BORLAND® C++

PROGRAMMIER-
HANDBUCH

BORLAND

ObjectWindows für C++

Borland C++ 3.0

Programmierhandbuch

BORLAND

Copyright © 1991 by Borland International, INC.
Jede Vervielfältigung dieses Handbuches sowie des Softwareprogrammes Borland C++ wird strafrechtlich verfolgt. Die Rechte an der Dokumentation und die Rechte an dem Softwareprogramm Borland C++ liegen bei Borland International.
Der rechtmäßige Erwerb der Programmdisketten und der Handbücher erlaubt die Nutzung der Programme analog der Benutzung eines Buches. Entsprechend der Unmöglichkeit, daß ein Buch zugleich an verschiedenen Orten von mehreren Personen gelesen wird, darf das Softwareprogramm Borland C++ nicht gleichzeitig von verschiedenen Personen an verschiedenen Orten und auf verschiedenen Geräten benutzt werden. Diskettenkopien dürfen lediglich zum Zweck der Datensicherung angefertigt werden.
Sie (der Lizenznehmer von Borland C++) dürfen eigene Programme, die Sie mit dem Compiler Borland C++ geschrieben und compiliert haben, ohne Einschränkung oder Verpiflichtung zu zusätzlichen Lizenzgebühren verwenden, weitergeben oder verkaufen, solange diese Programme einen Copyright-Vermerk enthalten. Unter Copyright-Vermerk verstehen wir entweder Ihren eigenen Copyright-Vermerk oder gegebenenfalls einen Hinweis wie »Created using Borland C++ by Borland International 1987, 88, 89, 90, 91«. Die Originaldisketten von Borland C++ beinhalten einige Dateien mit Hardware- und Zeichensatzinformationen, die von der Standard-Grafikbibliothek (GRAPHICS.LIB) verwendet werden. Diese Dateien mit den Suffixen CHR und BGI unterliegen besonderen Nutzungsbeschränkungen. Sie dürfen nicht als Dateien weiterverkauft werden. Sie dürfen diese Dateien aber in Ihren eigenen mit Borland C++ geschriebenen und compilierten Programmen verwenden. Sie dürfen diese compilierten Programme auch weiterverkaufen, solange sie einen Copyright-Vermerk tragen.
Die Beispielprogramme von Borland C++, die auf den Originaldisketten mitgeliefert werden, demonstrieren die verschiedenen Eigenschaften von Borland C++. Sie dienen ausschließlich zu Lehrzwecken. Sie dürfen diese Beispielprogramme editieren oder für den eigenen Gebrauch modifizieren. Sie dürfen diese Programme jedoch nicht als Ganzes oder in Teilen in compilierter Form oder als Quelltext weitergeben oder verkaufen. Sie dürfen die Routinen aus den Beispielprogrammen in eigenen Anwenderprogrammen verwenden, solange die resultierenden Programme nicht einem Beispielprogramm oder einem Teil eines Beispielprogrammes in Erscheinung oder Funktion gleichen.

Einschränkung der Gewährleistung
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber
Borland GmbH
Gautinger Straße 10
Postfach
8130 Starnberg
Telefon 0 18 51 / 2 63 - 0
Telefax 081 51 / 142 09

Übertragung ins Deutsche: Peter Baier, Roland Konstanty, Waltraud Kopelent, Arne Schäpers

Typographische Konzeption des Innenteils: Rudolf Paulus Gorbach

Grafiken: Andreas Höhne, Sid Gastl

Satz: Hanna Bergbauer

PostScript-Belichtung: CD Computer & Dokumentations GmbH, Neuler

Druck und Bindung: Schoder Druck GmbH, Gersthofen

Papier: Schleipen GmbH. Bad Dürkheim. Das verwendete Papier ist säurefrei und alterungsbeständig. Die Produktion erfolgt in einem geschlossenen Wasserkreislauf ohne Einsatz von optischen Aufhellern und unter Verwendung von chlorarm gebleichten, holzfreien Zellstoffqualitäten. Die Abwasserreinigung wird in einer betriebseigenen chemisch-biologischen Kläranlage vorgenommen.

2., unveränderte Auflage, Mai 1992

Inhalt

Einführung 1

Inhalt dieses Handbuchs 1
Hinweise zur formalen Sprachbeschreibung 2
Syntax und Terminologie 3

1. Lexikalische Grammatik 5

Whitespace 5
Zeilen mit \ verbinden 6
Kommentare 6
Symbole 8
Schlüsselwörter 8
Bezeichner 9
Konstanten 10
Integerkonstanten 12
Zeichenkonstanten 14
Gleitkommakonstanten 16
Aufzählungstyp-Konstanten 17
Stringkonstanten 17
Konstanten und ihre interne Darstellung 18
Interpunktionszeichen 20

2. Semantische Grammatik 25

Deklarationen 25
Objekte 25
L-Werte 26
Typen und Speicherklassen 27
Gültigkeitsbereich 27
Sichtbarkeit 28
Dauer 29
Übersetzungseinheiten 30
Bindung 31
Ergänzen von Namen 32
Deklarationssyntax 33
Tentative Definitionen 33
Mögliche Deklarationen 34

Externe Deklarationen und Definitionen 38
Typ-Spezifizierer 40
Typ-Taxonomie 40
Die grundlegenden Typen 41
Initialisierung 44
Einfache Deklarationen 46
Speicherklassen-Spezifizierer 46
Modifizierer 48
Komplexe Deklarationen und Deklaratoren 53
Zeiger 54
Zeiger auf Objekte 55
Zeiger auf Funktionen 55
Zeigerdeklarationen 56
Zeiger und Konstanten 57
Zeigerarithmetik 58
Arrays 59
Zeigerkonvertierungen 59
C++ Referenzdeklarationen 59
Funktionen 60
Deklarationen und Definitionen 60
Deklarationen und Prototypen 61
Definitionen 63
Deklaration formaler Parameter 64
Strukturen 65
Funktionsaufrufe und Argumentkonvertierungen 65
Strukturen und Typdefinitionen ohne »Tag«-Name 66
Strukturelement-Deklarationen 67
Strukturen und Funktionen 67
Zugriff auf die Strukturelemente 67
Ausrichtung auf Wortgrenzen in Strukturen 69
Namensbereich von Strukturen 69
Unvollständige Deklarationen 70
Bitfelder 70
Varianten 71
Anonyme Varianten (C++) 72
Aufzählungstypen 73
Deklarationen von Varianten 73
Ausdrücke 75
Ausdrücke in C++ 79
Auswertungsreihenfolge 79
Operatoren-Semantik 80
Fehler und Überläufe 80
Beschreibung der Operatoren 81
Unäre Operatoren 82
Binäre Operatoren 82
Suffix- und Präfix-Operatoren 83
Inkrement- und Dekrement-Operatoren 85

Unäre Operatoren 85
Der sizeof-Operator 87
Multiplikative Operatoren 88
Additive Operatoren 89
Bitweise Schiebeoperatoren 89
Relationale Operatoren 90
Gleichheitsoperatoren 91
Der bitweise UND-Operator & 92
Der bitweise Exklusives-ODER-Operator ^ 93
Der bitweise Inklusives-ODER-Operator | 93
Der logische UND-Operator && 93
Der logische ODER-Operator || 94
Der Bedingungsoperator ?: 94
Zuweisungs-Operatoren 95
Der Komma-Operator 96
C++ Operatoren 96
Anweisungen 97
Blockanweisungen 98
Sprungziele 99
Ausdruckanweisungen 99
Auswahlanweisungen 99
Wiederholungsanweisungen 101
Sprunganweisungen 103

3. C++ 105

Referenzen 105
Einfache Referenzen 105
Referenzen als Argumente 106
Der Gültigkeitsbereich-Der Gültigkeitsbereich-Zugriffsoperator 107
Die Operatoren new und delete 108
Fehlerbehandlung 109
Der Operator new im Zusammenhang mit Arrays 109
Der Operator delete im Zusammenhang mit Arrays 109
Der ::operator new 110
Initialisierer in Zusammenhang mit dem Operator new 110
Klassen 111
Klassennamen 111
Klassentypen 111
Der Gültigkeitsbereich von Klassennamen 112
Klassenobjekte 113
Die Klassenelementliste 113
Elementfunktionen 113
Das Schlüsselwort this 113
Inline-Funktionen 114
Statische Elemente 115
Der Gültigkeitsbereich von Elementen 116

Der Zugriff auf Basisklassen und abgeleitete Klassen 119
Virtuelle Basisklassen 122
Friends von Klassen 122
Konstruktoren und Destruktoren 124
Konstruktoren 125
Der Standardkonstruktor 126
Der Kopierkonstruktor 126
Überladen von Konstruktoren 127
Aufrufreihenfolge von Konstruktoren 127
Die Initialisierung von Klassen 129
Destruktoren 131
Wann werden Destruktoren aufgerufen? 132
atexit, #pragma exit und Destruktoren 132
exit und Destruktoren 132
abort und Destruktoren 132
Virtuelle Destruktoren 133
Überladene Operatoren 134
Operatorfunktionen 135
Die »Vererbung« von überladenen Operatoren 136
Überladen von new und delete 136
Das Überladen von unären Operatoren 137
Das Überladen von binären Operatoren 138
Überladen des Zuweisungsoperators = 138
Überladen des Funktionsaufruf-Operators () 139
Überladen des 139
Überladen des Operators -> 139
Virtuelle Funktionen 140
Abstrakte Klassen 141
Der Gültigkeits Der Gültigkeitsbereich in C++ 142
Gültigkeitsbereich Klasse 143
Verbergen 143
Zusammenfassung der C++ Gültigkeitsbereichsregeln 143
Templates 144
Funktions-Templates 145
Klassen-Templates 148

4. Der Präprozessor 153

Die Null-Direktive # 155
Die Direktiven #define und #undef 155
Einfache #define-Makros 156
Die Direktive #undef 156
Die Option Definitionen 158
Schlüsselwörter und geschützte Wörter 158
Makros mit Parametern 158
Dateien mit #include einbinden 161
Header-Dateien suchen mit <Header_Name> 162

Header-Dateien suchen mit "Header_Name" 162
Bedingte Compilierung 163
Bedingungsdirektiven: #if, #elif, #else und #endif 163
Die bedingten Direktiven #ifdef und #ifndef 164
Die Direktive #line 165
Die Direktive #error 166
Die Direktive #pragma 166
#pragma argsused 167
#pragma exit und #pragma startup 167
#pragma hdrfile 168
#pragma hdrstop 168
#pragma saveregs 168
Vordefinierte Makros 169
__CDECL__ 169
__cplusplus 169
__DATE__ 169
__DLL__ 170
__FILE__ 170
__LINE__ 170
__MSDOS__ 170
__PASCAL__ 170
__STDC__ 170
__TCPLUSPLUS__ 171
__TEMPLATES__ 171
__TIME__ 171
__TURBOC__ 171
_Windows 171

5. C++ Streams 173

Was ist ein Stream? 173
Die I/O-Stream-Die I/O-Stream-Bibliothek 173
Ausgabe 175
Eingabe 179
Ein-/Ausgabe von benutzerdefinierten Typen 180
Einfache Datei-Einfache Datei-Ein-/Ausgabe 181
String-Streams 182
Stream-Klassenreferenz 183

6. Arithmetik 201

Gleitkomma-Optionen 201
Die Emulation des 80x87 201
Verwendung des 80x87-Codes 202
Falls Sie keine Gleitkomma-Operationen brauchen 202
Beschleunigte Gleitkommaberechnungen 202
Register und Koprozessor 203

Fehlerbedingungen bei Gleitkomma-Operationen 203
Komplexe Arithmetik 204
Die BCD-Arithmetik 205

7. BASM und Inline-Assembler 207

Inline-Assemblierung 207
BASM 207
Inline-Syntax 208
Opcodes 209
Der Zugriff auf Daten und Funktionen 211
Zugriffe auf Strukturelemente 212
Interrupt-Funktionen 213
Sprungbefehle und Labels 213
Low-Level-Praktiken am Beispiel 214

8. Erstellen einer Windows-Anwendung 217

Compilieren und Linken in der IDE 217
Die Ressourcendateien 219
Die Moduldefinitionsdatei 219
Compilieren und Linken von WHELLO 219
WinMain 221
Prologe und Epiloge 221
Windows: alle Funktionen exportierbar 222
Windows: explizite Funktionen exportiert 222
Windows: intelligente Callbacks 223
Windows DLL: alle Funktionen exportierbar 223
Windows DLL: explizit exportierte Funktionen 223
Das Schlüsselwort _export 224
Zusammenfassung: Prologe, Epiloge und Exporte 224
Speichermodelle 225
Moduldefinitionsdateien 225
Ein kurzes Beispiel 226
Linken für Windows 227
Linken in der IDE 227
Dynamische Link Dynamische Linkbibliotheken (DLL) 228
Compilieren und Linken einer dynamischen Linkbibliothek in der IDE 228
Anlegen einer Importbibliothek 229
Anlegen von dynamischen Linkbibliotheken 230

9. Die ANSI-C-Implementation bei Turbo C++ 235

10. Fehlermeldungen 247

Auffinden einer Fehlermeldung 247
Arten von Fehlermeldungen 248
Compiler-Meldungen 248
Hilfe-Compiler-Meldungen 249
Laufzeit-Meldungen 250
Meldungen der Bibliotheksverwaltung 251
Linker-Meldungen 251
Erläuterung der Meldungen 252

A. HC: Der Hilfe-Compiler von Windows 333

Das Hilfesystem anlegen: Der Entwicklungszyklus 333
Das Hilfesystem und der Anwender 334
Das Hilfesystem und der Autor 335
Das Hilfesystem und der Programmierer 335
Das Hilfesystem planen 336
Einen Plan entwickeln 336
Themendateistruktur festlegen 339
Hilfethemen auf dem Bildschirm darstellen 342
Themendateien erstellen 344
Grafikbilder 344
Geeignetes Textprogramm auswählen 345
Themendateien strukturieren 345
Themendateien codieren 346
Grafikbilder einbringen 354
Themendateien verwalten 356
Die Hilfedatei compilieren 358
Hilfe-Projektdatei erstellen 359
Themendateien festlegen: 360
Symbolische Namen festlegen: 360
Optionen festlegen: 361
Alternative Kontextstrings angeben: 366
Kontextabhängige Themen anpassen: 367
Referenzierte Bitmaps: 369
Hilfedateien compilieren 369
Schnittstelle zum Hilfesystem programmieren 371
Hilfe-Beispiele 377
Die Projektdatei für Helpex 379

B. Turbo Editor Macro Language (TEML) 381

TEMC-Kommandozeile 381
Syntax 382
Tastenschlüssel 383
Tastenbenennungen 383
Vordefinierte Editorbefehle 384

Transfermakros 391
Statusmakros 391
Dateinamenmakros 391
Anweisungsmakros 391
Aufrufen von DOS-Kommandos 396
Einstellung des Transferspeichers 396

Index 397

Einführung

Dieses Handbuch richtet sich an erfahrene Programmierer. Wenn Sie sehr gut programmieren (in C, C++ oder einer anderen Sprache), dann ist dieses Handbuch für Sie das richtige. Es enthält eine Sprachreferenz, Informationen über C++ Streams, den Präprozessor, Gleitkommaberechnung, BASM, Inline-Assembler, ANSI-Implementierung, Fehlermeldungen, den Windows-Hilfecompiler und die Editor-Makrosprache TEML.

Die Programmbeispiele enthalten eine *main*-Funktion. *EasyWin* macht alle Beispiele unter Windows lauffähig, so daß Sie *WinMain* nicht mit ihren komplizierten Parametern einsetzen müssen.

In der »Einführung« und in Kapitel 1 »Die integrierte Entwicklungsumgebung« des *Benutzerhandbuchs* finden Sie einen Überblick über die Turbo C++ Dokumentation und Hinweise dazu, wie Sie die Handbücher effektiv zur Ihrer Unterstützung heranziehen sollten.

Beginnen Sie mit der Lektüre des *Benutzerhandbuchs*, wenn eine der folgendenen Bedingungen auf Sie zutrifft:

1. Wenn Sie noch nie programmiert haben;

2. wenn Sie zwar programmieren können, aber weder C, C++ noch Windows kennen und sich deshalb erst einen Überblick verschaffen möchten;

3. wenn Sie Informationen zur der Installation von Turbo C++ benötigen;

4. wenn Sie sich über die integrierte Entwicklungsumgebung (IDE) von Turbo C++ (einschließlich des dazugehörigen Editors), die Programmverwaltung, die Konvertierung von Microsoft-C-Programmen und die Verwendung von EasyWin (für den Einsatz von DOS-Programmen) unterrichten wollen.

Im *Benutzerhandbuch* finden Sie außerdem eine Erläuterung der Typographie und der verwendeten Symbole.

Inhalt dieses Handbuchs

Die Kapitel 1 »Lexikalische Grammatik«, 2 »Semantische Grammatik«, 3 »C++« und 4 »Der Präprozessor« beschreiben Turbo C++, wobei alle Unterschiede zum ANSI-C-Standard vermerkt sind. Des weiteren finden Sie hier eine formale

Beschreibung der Sprache, eine Sprachreferenz und die Syntax von C und C++.Im nächsten Abschnitt dieser Einführung wird ausführlicher auf diese Kapitel eingegangen.

Kapitel 5 »C++Streams« erläutert den Einsatz der C++Stream-Bibliothek Version 2.1.

Kapitel 6 »Arithmetik« befaßt sich mit Gleitkomma- und BCD-Arithmetik.

Kapitel 6 »Die Container-Klassenbibliotheken« beschreibt die Verwendung der Containerklassen (einschließlich der Vorlagen).

Kapitel 7 »BASM und Inline-Assembler« erklärt Ihnen, wie Sie Assemblerprogramme erstellen sollten, damit diese beim Aufruf von Turbo C++Programmen aus fehlerfrei laufen. Das Kapitel erläutert außerdem den integrierten Assembler in der IDE.

Kapitel 8 »Erstellen einer MS-Windows-Anwendung« liefert Ihnen eine Einführung in die Windows-Programmierung.

Anhang 9 »Spezielle ANSI-C-Implementierungsstandards« beschreibt Aspekte des ANSI-C-Standards, die nur unzureichend oder gar nicht definiert sind. Diese Punkte können sich von Implementierung zu Implementierung unterscheiden. In diesem Kapitel wird beschrieben, wie Turbo C++ im Hinblick auf diese Aspekte funktioniert.

Kapitel 10 »Fehlermeldungen« enthält alle Fehler und Warnungen zusammen mit einem Lösungsvorschlag. Diese Laufzeit-, Compiler-, Bibliotheksverwaltungs- und Hilfecompiler-Fehler sind alphabetisch angeordnet.

Anhang A »HC: Der Hilfe-Compiler von Windows« stellt Ihnen den Einsatz dieses Compilers für die Erstellung von Hilfesystemen für Windows-Programme vor.

Anhang B »Turbo Editor Macro Language (TEML)« beschreibt die Makrosprache des Turbo-Editors.

Hinweise zur formalen Sprachbeschreibung

Kapitel 1 bis 4 beinhalten eine formale Beschreibung der Sprache, eine Sprachreferenz und die Syntax von C und C++. Diese Kapitel beschreiben die Sprache Turbo C++allgemein sowie ihre Implementierung duch Turbo C++. Die Kapitel sind allerdings nicht als C-Lehrbuch zu verstehen. Zur Darstellung der Syntax verwenden wir eine modifizierte Backus-Naur-Notation und fügen – wenn es angebracht erscheint – kurze Erläuterungen und Programmbeispiele hinzu.

Die Kapitel sind folgendermaßen gegliedert:

Kapitel 1 »Lexikalische Grammatik« befaßt sich mit den wortähnlichen Einheiten einer Sprache, den sogenannten »Tokens«. Lexikalische Elemente gehören

zu den verschiedenen Kategorien wortähnlicher Einheiten (den »Tokens«), die von einer Sprache erkannt werden.

Kapitel 2 »Semantische Grammatik« erläutert, wie die Elemente von Turbo C++ verwendet werden. Die semantische Grammatik bestimmt die Regeln, nach denen Symbole zur Bildung von Ausdrücken, Anweisungen oder anderen Einheiten kombiniert werden können.

Kapitel 3 »C++« beschreibt die C++ spezifischen Aspekte von Turbo C++.

Kapitel 4 »Der Präprozessor« befaßt sich mit dem Präprozessor und seinen Auswirkungen auf Makros, Include-Dateien und Pragmas.

Turbo C++ verfügt über alle Eigenschaften der AT&T C++ Version 2.1. C++ wurde von Bjarne Stroustrup von AT&T Bell Laboratories entwickelt und erweitert C um objektorientierte Sprachelemente. C++ verfügt über zahlreiche neue Merkmale und Fähigkeiten, unterscheidet sich aber in vielen Punkten mehr oder weniger von C. Auf diese Unterschiede wird in den genannten Kapiteln explizit hingewiesen. Alle Turbo C++ Spracheigenschaften, die von C++ abgeleitet sind, werden in Kapitel 3 »C++« ausführlich erläutert.

Turbo C++ entspricht dem ANSI-C-Standard, enthält aber einige Ergänzungen, auf die im Text gesondert hingewiesen wird. Mit Hilfe entsprechender Parameter können Sie festlegen, ob der Compiler beim Antreffen der Ergänzungen eine Warnung ausgibt oder ob hinzugekommene Schlüsselwörter als normale Bezeichner behandelt werden.

Außerdem gibt es »konforme« Ergänzungen, die über die #*pragma*-Direktiven zur Verfügung gestellt werden. Die #*pragma*-Direktiven von ANSI C dienen zur Behandlung nicht-standardmäßiger, implementationsabhängiger Funktionen.

Syntax und Terminologie

Syntaktische Definitionen setzen sich aus dem Namen der zu definierenden Nicht-Terminale und einem Doppelpunkt (:) zusammen. Alternativen werden normalerweise in nachfolgenden Einzelzeilen angegeben, können aber auch in eine einzige Zeile geschrieben werden, wenn der Ausdruck *Eines der ...* vorangestellt wird, zum Beispiel:

externe Definition:
    Funktionsdefinition
    Deklaration

Oktalziffer: eines der folgenden Zeichen:
0 1 2 3 4 5 6 7

Optionale Elemente eines Konstrukts werden in spitzen Klammern dargestellt:

Integer-Suffix:
    unsigned-Suffix <*long-Suffix*>

Im vorliegenden Kapitel verstehen wir unter einem »Argument« den tatsächlichen Wert, der dem Aufruf einer Funktion übergeben wird. Ein »Parameter« ist die im Funktionskopf definierte Variable, die den Wert erhält.

1. Lexikalische Grammatik

Dieses Kapitel enthält eine formale Definition der lexikalischen Grammatik von Turbo C++. Diese befaßt sich mit den wortähnlichen Einheiten einer Sprache, sogenannten »Symbolen« oder »Tokens«. Die semantische Grammatik (siehe Kapitel 2 »Semantische Grammatik«) bestimmt die Regeln, nach denen Symbole zur Bildung von Ausdrücken, Anweisungen oder anderen Einheiten kombiniert werden können.

In Turbo C++ ergeben sich die Symbole als Folge der Operationen, die der Compiler und sein Präprozessor mit Ihrem Programm durchführen.

Ein Turbo C++ Programm ist eine Abfolge von ASCII-Zeichen, die den Quellcode darstellen, der mit einem Texteditor (zum Beispiel mit dem Turbo C++ Editor) erstellt wurde. Die grundlegende Programmeinheit in Turbo C++ ist die Datei. Sie entspricht einer benannten DOS-Datei im Speicher oder auf Platte und hat die Namenserweiterung .C oder .CPP.

Der Präprozessor durchsucht die Programme zuerst nach Präprozessor-Direktiven (vgl. Kapitel 4 »Der Präprozessor«). So fügt beispielsweise die Direktive *#include <inc_file>* vor der Compilierung den Inhalt der Datei INC_FILE in das Programm ein. Der Präprozessor erweitert auch Makros, die im Programm und in Include-Dateien stehen.

Whitespace

In der lexikalischen Analysephase der Compilierung wird die Quellcodedatei in Symbole und »Whitespace« geparst (d.h. zerlegt). *Whitespace* ist der Sammelbegriff für Zeichen, die als Trenner gelten: Leerzeichen, Tabulatoren (horizontale und vertikale), Zeilenvorschübe und Kommentare. Whitespace dient zur Markierung von Beginn und Ende eines Symbols, aber abgesehen davon wird Whitespace ignoriert.

Die folgenden beiden Sequenzen:

```
int   i; float  f;
```

und

```
int  i ;
     float   f;
```

sind lexikalisch gesehen identisch und werden in dieselben sechs Symbole zerlegt:

```
1. int
2. i
3. ;
4. float
5. f
6. ;
```

Wenn als Whitespace fungierende ASCII-Zeichen innerhalb *literaler* (*numerischer* und *alphanumerischer*) *Strings* stehen, werden sie vom normalen Parsingvorgang ausgenommen, sie bleiben daher als Teil des Strings erhalten:

```
char name[] = "Borland International";
```

wird in sieben Symbole zerlegt, einschließlich des als Einzelsymbol angesehenen "Borland International".

Zeilen mit \ verbinden

Ein Sonderfall ist das Auftreten eines Backslash (\) vor dem Zeilenvorschubzeichen. Der Backslash und der Zeilenvorschub werden ignoriert, mit der Konsequenz, daß zwei physikalisch vorhandene Zeilen als eine Einheit betrachtet werden. Aus

```
"Borland \
International"
```

wird "Borland International" (vergleichen Sie hierzu den Abschnitt »Stringkonstanten« weiter hinten in diesem Kapitel).

Kommentare

Kommentare sind Textteile, die Erklärungen zum Programm enthalten. Sie sind nur für den Programmierer von Nutzen und werden vor dem Parsen aus dem Quelltext entfernt.

Es gibt zwei Arten, Kommentare auszuweisen: die traditionelle C- und die C++ Methode. Beide werden von Turbo C++ unterstützt, mit der optionalen Erweiterung von verschachtelten Kommentaren. Sie können beide Methoden in C und C++ Programmen einsetzen.

C-Kommentare

Ein C-Kommentar ist eine Zeichenfolge, die nach dem Zeichenpaar /* steht. Der Kommentar endet bei dem ersten Zeichenpaar */, das auf das Startsymbol /* folgt. Die Kommentarbegrenzungszeichen und die von ihnen eingeschlossene Zeichenfolge werden *nach* der Makroerweiterung durch ein Leerzeichen ersetzt. Beachten Sie, daß manche C-Implementierungen Kommentare entfernen, ohne sie durch Leerzeichen zu ersetzen.

In Turbo C++ ist es nicht möglich, Symbole mit Hilfe von /* */ zu verschmelzen (diese Methode ist ohnehin nicht portabel). Das sogenannte »token pasting« ist ANSI-C-konform mit dem Symbolpaar ## implementiert:

```
#define VAR(i,j)   (i/**/j)       /* unzulässig */
#define VAR(i,j)   (i##j)         /* zulässig in Turbo C++ */
#define Var(i,j)   (i ## j)       /* ebenfalls zulässig */
```

In Turbo C++ wird

```
int /* Deklaration */ i /* Zähler */;
```

zerlegt in

```
int   i   ;
```

und ergibt somit die drei Symbole *int i ;*.

Ausführliche Informationen über das Ersetzen von Symbolen finden Sie in Kapitel 4 »Der Präprozessor« im Abschnitt »Makros mit Parametern«.

Verschachtelte Kommentare

In ANSI C sind verschachtelte Kommentare nicht zulässig. Wenn Sie die vorausgehende Zeile folgendermaßen herauskommentieren wollen:

```
/*  int /* Deklaration */ i /* Zähler */;  */
```

tritt ein Fehler auf, weil der Bereich des ersten /* am ersten */ endet. Dies ergibt

```
i ; */
```

wodurch ein Syntaxfehler verursacht wird.

Per Voreinstellung verbietet Turbo C++ verschachtelte Kommentare. Diese Einstellung kann jedoch mit Hilfe von Compileroptionen geändert werden. Sie können verschachtelte Kommentare über das Dialogfenster des Menüs *Optionen/Compiler/Quellcode* in der integrierten Entwicklungsumgebung ermöglichen.

C++ Kommentare

In C++ können Sie einen einzeiligen Kommentar mit zwei Schrägstrichen (//) anlegen. Der Kommentar kann an beliebiger Stelle beginnen und erstreckt sich bis zur nächsten neuen Zeile:

```
class X {  // Kommentar
... };
```

Hinweis: Auch in reinem C-Code können mit zwei Schrägstrichen Kommentare erstellt werden. Dies funktioniert allerdings nur in Turbo C++.

Kommentarbegrenzer und Whitespace

In seltenen Fällen können durch Verwendung von (syntaktisch nicht erforderlichen) Whitespace vor /* bzw. // und nach */ Portabilitätsprobleme vermieden werden. Beispielsweise wird der C++ Code

```
int i = j//* dividiert durch k*/k;
+m;
```

zerlegt in

```
int i = j +m;
```

nicht in

```
int i = j/k;
+m;
```

wie es normalerweise in C der Fall wäre. Dieses Problem läßt sich durch die folgende, besser lesbare Schreibweise vermeiden:

```
int i = j/ /* dividiert durch k*/ k;
+m;
```

Symbole

Turbo C++ kennt folgende sechs Arten von *Symbolen* :

Symbol:
- *Schlüsselwort*
- *Bezeichner*
- *Konstante*
- *String-Literal*
- *Operator*
- *Interpunktionszeichen*

Interpunktionszeichen werden manchmal auch als Trennzeichen bezeichnet. Beim Parsen des Quellcodes wird aus einer gegebenen Zeichenfolge zuerst das längstmögliche Symbol gebildet. Dadurch wird etwa *external* als Bezeichner ermittelt und nicht in das Schlüsselwort **extern** und den Bezeichner *al* zerlegt.

Schlüssel-wörter

Schlüsselwörter sind für spezielle Zwecke reservierte Wörter, die nicht als normale Bezeichnernamen verwendet werden dürfen. In den folgenden Tabellen sind alle Turbo C++ Schlüsselwörter aufgelistet. Mit Kommandozeilenparametern oder den Optionen der integrierten Entwicklungsumgebung können Sie die Auswahl auf ANSI-Schlüsselwörter, UNIX-Schlüsselwörter etc. beschränken. Näheres hierzu finden Sie in Kapitel 1 »Die integrierte Entwicklungsumgebung« des *Benutzerhandbuchs*.

Tabelle 1.1 Alle Turbo C++ Schlüsselwörter

_asm	case	class	default
asm	_cdecl	const	delete
auto	cdecl	continue	do
break	char	_cs	double

_ds	huge	pascal	struct
else	if	private	switch
enum	inline	protected	template
_es	int	public	this
_export	_interrupt	register	typedef
extern	interrupt	return	union
_far	_loadds	_saveregs	unsigned
far	long	_seg	virtual
float	_near	short	void
for	near	signed	volatile
friend	new	sizeof	while
goto	operator	_ss	
_huge	_pascal	static	

Tabelle 1.2 Turbo C++ Schlüsselwörter (als Erweiterungen zu ANSI C)

_cdecl	_export	_loadds	_saveregs
cdecl	_far	_near	_seg
_cs	far	near	_ss
_ds	huge	_pascal	
_es	interrupt	pascal	

Tabelle 1.3 C++ Schlüsselwörter

asm	inline	protected	virtual
class	new	public	
delete	operator	template	
friend	private	this	

Tabelle 1.4 Register-Pseudovariablen in Turbo C++

_AH	_BP	_CX	_DX	_SS
_AL	_BX	_DH	_ES	
_AX	_CH	_DI	_FLAGS	
_BH	_CL	_DL	_SI	
_BL	_CS	_DS	_SP	

Bezeichner

Bezeichner können aus folgenden Einheiten bestehen:

Bezeichner:
Nicht-Ziffer
Bezeichner Nicht-Ziffer
Bezeichner Ziffer

Nicht-Ziffer: eines der folgenden Zeichen:
a b c d e f g h i j k l m n o p q r s t u v w x y z _
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Ziffer: eines der folgenden Zeichen:
0 1 2 3 4 5 6 7 8 9

Namen- und Längenbeschränkung

Bezeichner sind beliebige Namen von beliebiger Länge für Klassen, Objekte, Funktionen, Variablen, benutzerdefinierte Datentypen, etc. Bezeichner können die Buchstaben A bis Z, a bis z, den Unterstrich (_) und die Ziffern 0 bis 9 enthalten. Es gibt jedoch zwei Einschränkungen:

1. Das erste Zeichen muß ein Buchstabe oder der Unterstrich sein.

2. Per Voreinstellung sind nur die ersten 32 Zeichen signifikant. Die Anzahl der signifikanten Zeichen kann durch Menüpunkte und Kommandozeilenparameter zwar *reduziert*, aber nicht erhöht werden. Verwenden Sie hierzu die Option *Bezeichnerlänge* des Menüs *Optionen/Compiler/Quellcode* in der integrierten Entwicklungsumgebung, wobei 1 <=n <=32 ist. Beachten Sie, daß in C++Programmen keine Beschränkung für die Anzahl signifikanter Zeichen besteht.

Bezeichner und Groß- und Kleinschreibung

In Turbo C++ wird zwischen Groß- und Kleinschreibung unterschieden, so daß *Sum*, *sum* und *suM* unterschiedliche Bezeichner sind.

Für globale Bezeichner aus anderen Modulen gelten dieselben Regeln in Bezug auf Namensgebung und signifikante Zeichenanzahl wie für normale Bezeichner. Die Berücksichtigung der Groß- und Kleinschreibung kann in Turbo C++ jedoch außer Kraft gesetzt werden. Über *Beim Linken Groß-/Kleinschreibung beachten* im Dialogfenster *Linker*(*Optionen/Linker/Einstellungen*) können Sie für globale Bezeichner die Berücksichtigung der Groß- und Kleinschreibung ausschalten. Unter dieser Bedingung werden *Sum* und *sum* als identische Bezeichner betrachtet. Dies kann beim Linken allerdings die Warnung `Duplicate symbol` zur Folge haben.

Eine Ausnahme von dieser Regel bilden Bezeichner vom Typ **pascal**, die beim Linken immer in Großbuchstaben umgewandelt werden.

Eindeutigkeit und Gültigkeitsbereich von Bezeichnern

Bezeichner können beliebige Namen sein, die den geltenden Regeln entsprechen. Es kann jedoch zu Fehlern kommen, wenn derselbe Name innerhalb desselben Gültigkeitsbereichs für mehrere Bezeichner verwendet wird, die denselben *Namensbereich* haben. Gleiche Namen sind für verschiedene Namensbereiche zulässig, unabhängig vom Gültigkeitsbereich.

Eine Beschreibung der Namensregelung finden Sie im Abschnitt »Gültigkeitsbereich« in Kapitel 2 »Semantische Grammatik«.

Konstanten

Konstanten sind Symbole, die für feste numerische oder Zeichenwerte stehen. Turbo C++ kennt vier Klassen von Konstanten: Gleitkomma, Integer, Aufzählung und Zeichen. Der Datentyp einer Konstante wird vom Compiler auf Grund ihres numerischen Wertes und ihres Formats im Quelltext abgeleitet.

Tabelle 1.5 zeigt die formale Definition einer Konstante.

Tabelle 1.5 Konstanten – formale Definition

Konstante:
Gleitkommakonstante
Integerkonstante
Aufzählungstyp-Konstante
Zeichenkonstante
Gleitkommakonstante:
Fraktionale-Konstante <Exponent><Gleitkomma-Suffix>
Ziffernfolge Exponent <Gleitkomma-Suffix>

Fraktionale-Konstante:
<Ziffernfolge> . Ziffernfolge
Ziffernfolge .

Exponent:
e <Vorzeichen>Ziffernfolge
E <Vorzeichen>Ziffernfolge

Vorzeichen: eines der folgenden Zeichen:
+ -

Ziffernfolge:
Ziffer
Ziffernfolge Ziffer

Gleitkomma-Suffix: eines der folgenden Zeichen:
f l F L

Integerkonstante:
Dezimale-Konstante <Integer-Suffix>
Oktale-Konstante <Integer-Suffix>
Hexadezimale-Konstante <Integer-Suffix>

Dezimale-Konstante:
Nicht-Null-Ziffer
Dezimale-Konstante Ziffer

Oktale-Konstante:
0
Oktale-Konstante Oktalziffer

Hexadezimale-Konstante:
0 x Hexziffer
0 X Hexziffer
Hexadezimale-Konstante Hexziffer

Nicht-Null-Ziffer: eines der folgenden Zeichen:
1 2 3 4 5 6 7 8 9

Oktalziffer: eines der folgenden Zeichen:
0 1 2 3 4 5 6 7

Hexziffer: eines der folgenden Zeichen:
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

Integer-Suffix
unsigned-Suffix <*long-Suffix*>
long-Suffix <*unsigned-Suffix*>

unsigned-Suffix: eines der folgenden Zeichen:
u U

long-Suffix: eines der folgenden Zeichen:
l L

Aufzählungstyp-Konstante:
Bezeichner

Zeichenkonstante:
c-Zeichenfolge

c-Zeichenfolge:
c-Zeichen
c-Zeichenfolge c-Zeichen

c-Zeichen
Jedes Zeichen außer dem Hochkomma ('), *dem Backslash (\)* *oder einem Zeilenvorschub (Escape-Sequenz).*

Escape-Sequenz: eine der folgenden Zeichenfolgen:

\"	\'	\?	\\
\a	\b	\f	\n
\o	\oo	\ooo	\r
\t	\v	\Xh...	\xh...

Integer-konstanten

Integerkonstanten können dezimale (Basis 10), oktale (Basis 8) oder hexadezimale (Basis 16) Zahlen sein. Sind keine Suffixe vorhanden, wird der Datentyp einer Integerkonstanten von ihrem Wert abgeleitet (vergleichen Sie Tabelle 1.6 weiter unten).

Beachten Sie, daß für dezimale und nichtdezimale Konstanten unterschiedliche Regeln gelten.

Dezimalkonstanten

Es sind *Dezimalkonstanten* von 0 bis 4294967295 zugelassen. Konstanten außerhalb dieses Bereichs liefern einen Fehler. Dezimalkonstanten dürfen nicht mit einer Null (0) beginnen, ansonsten werden sie als oktale Konstanten interpretiert:

```
int i = 10;    /*dezimal 10 */
int i = 010;   /*dezimal 8 */
int i = 0;     /*dezimal 0 = oktal 0 */
```

Oktale Konstanten

Alle Konstanten, die mit einer Null beginnen, werden als *oktale Konstanten* interpretiert. Enthält eine oktale Konstante die unzulässigen Ziffern 8 oder 9, wird ein Fehler gemeldet. Oktale Konstanten werden abgeschnitten, wenn sie größer als 037777777777 sind.

Hexadezimale Konstanten

Alle Konstanten, die mit 0x (oder 0X) beginnen, werden als hexadezimale Konstanten interpretiert. Hexadezimale Konstanten werden abgeschnitten, wenn sie größer als 0xFFFFFFFF sind.

long- und unsigned-Suffixe

Beinhaltet eine Konstante das Suffix *L* (oder *l*), wird sie als **long** dargestellt. Das Suffix *U* (oder *u*) bedingt eine **unsigned** (vorzeichenlose) Konstante. Eine Konstante ist **unsigned long**, wenn ihr Wert größer als dezimal 65535 ist, unabhängig von der Basis. Die Suffixe *L* und *U* können in beliebiger Reihenfolge und Schreibweise an eine Konstante angefügt werden: *ul*, *lu*, *UL*, etc.

Tabelle 1.6 faßt die Repräsentation von Integerkonstanten in allen drei Basen zusammen. Bei der Angabe der Datentypen wird davon ausgegangen, daß kein *L*- oder *U*-Suffix eingesetzt wurde.

Tabelle 1.6 Turbo C++ Integerkonstanten ohne- L- oder U-Suffix

Integerkonstanten	Datentypen
Dezimalkonstanten	
0 bis 32767	**int**
32768 bis 2147483647	**long**
2147483648 bis 4294967295	**unsigned long**
>4294967295	wird abgeschnitten.
Oktale Konstanten	
00 bis 077777	**int**
0100000 bis 0177777	**unsigned int**
02000000 bis 017777777777	**long**
020000000000 bis 037777777777	**unsigned long**
>037777777777	wird abgeschnitten.
Hexadezimale Konstanten	
0x0000 bis 0x7FFF	**int**
0x8000 bis 0xFFFF	**unsigned int**
0x10000 bis 0x7FFFFFFF	**long**
0x80000000 bis 0xFFFFFFFF	**unsigned long**
>0xFFFFFFFF	wird abgeschnitten.

Ist kein *U-*, *u-*, *L-* oder *l*-Suffix vorhanden, entspricht der Datentyp einer Konstanten dem ersten zu ihrem Wert passenden Typ:

dezimal	**int**, **long int**, **unsigned long int**
oktal	**int**, **unsigned int**, **long int**, **unsigned long int**
hexadezimal	**int**, **unsigned int**, **long int**, **unsigned long int**

Hat die Konstante das Suffix *U* oder *u*, entspricht der Datentyp dem ersten **unsigned int** bzw. **unsigned long int**, je nachdem, was zum Wert paßt.

Hat die Konstante das Suffix *L* oder *l*, entspricht der Datentyp dem ersten **long int** bzw. **unsigned long int**, je nachdem, was zum Wert paßt.

Hat die Konstante beide Suffixe (*ul, lu, lU, uL, Lu, LU* oder *UL*), ist sie von Datentyp **unsigned long int**.

Zeichen-konstanten

Eine *Zeichenkonstante* besteht aus einem oder mehreren in Apostrophen eingeschlossenen Zeichen, wie etwa 'A', '=', '\n'.

In C ist eine Zeichenkonstante vom Datentyp **int** und umfaßt 16 Bits, wobei das höherwertige Byte den Wert 0 oder das Vorzeichen enthält. In C++ hat eine Zeichenkonstante den Datentyp **char**. Konstanten mit mehreren Zeichen haben in C und C++ den Typ **int**.

Escape-Sequenzen

Mit dem Backslash (\) beginnt eine *Escape-Sequenz*, die zur Repräsentation nicht darstellbarer Zeichen dient (z.B. steht *\n* für das Zeichen »Zeilenvorschub«. Ein Backslash in Kombination mit oktalen oder hexadezimalen Zahlen repräsentiert den ASCII- oder Steuercode, der diesem Wert entspricht, z.B. *\03* für *Ctrl-C* oder *\x3F* für das Fragezeichen. Eine Escape-Sequenz kann einen String mit bis zu drei oktalen oder beliebig vielen hexadezimalen Ziffern umfassen, vorausgesetzt, der Wert liegt im zulässigen Bereich des Datentyps **char** (0 bis 0xff in Turbo C++). Größere Zahlen liefern den Compilerfehler `Numeric constant too large`. Die Oktalzahl \777 ist zum Beispiel größer als der zulässige Höchstwert \377 und ergibt somit einen Fehler. Das erste nicht-oktale oder nicht-hexadezimale Zeichen in einer oktalen oder hexadezimalen Escape-Sequenz markiert das Ende der Sequenz.

Ursprünglich ließ Turbo C nur drei Ziffern in einer hexadezimalen Escape-Sequenz zu. Die in Turbo C++ aufgenommenen ANSI-C-Regeln können Probleme bei älterem Programmcode ergeben, der davon ausgeht, daß nur die ersten drei Zeichen konvertiert werden. Wird zum Beispiel in der Turbo C Version 1.x ein String mit einem Alarmton (ASCII 7), gefolgt von numerischen Zeichen definiert, würde der Programmcode folgendermaßen aussehen:

```
printf ("\x0072.1A Simple Operating System");
```

Diese Zeile soll als *\x007* und *"2.1A Simple Operating System"* interpretiert werden. In Turbo C++ wird *\x0072* als hexadezimale Zahl und *".1A Simple Operating System"* als literaler String interpretiert.

Zur Umgehung derartiger Probleme sollte die Programmzeile folgendermaßen neu geschrieben werden:

```
printf ("\x007" "2.1A Simple Operation System");
```

Mehrdeutigkeiten entstehen auch, wenn auf eine oktale Escape-Sequenz eine nicht-oktale Ziffer folgt. Da 8 und 9 unzulässige oktale Ziffern sind, würde z.B. die Konstante \258 als Zwei-Zeichen-Konstante interpretiert, die sich aus den zwei Zeichen \25 und 8 zusammensetzt.

In der folgenden Tabelle sind alle Escape-Sequenzen zusammengestellt. Beachten Sie, daß zur Darstellung des normalen ASCII-Backslash (\), wie zum Beispiel in DOS-Pfaden, \\ verwendet werden muß.

Tabelle 1.7 Turbo C++ Escape-Sequenzen

Sequenz	Wert	Zeichen	Bedeutung
\a	0x07	BEL	Alarmton
\b	0x08	BS	Backspace
\f	0x0C	FF	Seitenvorschub
\n	0x0A	LF	Zeilenvorschub
\r	0x0D	CR	Wagenrücklauf (Carriage Return)
t	0x09	HT	Tabulator horizontal
\v	0x0B	VT	Tabulator vertikal
\\	0x5c	\	Backslash (umgekehrter Schrägstrich)
\'	0x27	'	Einfaches Anführungszeichen (Apostroph)
\"	0x22	"	Doppelte Anführungszeichen
\?	0x3F	?	Fragezeichen
\O		beliebig	O = ein String mit 1 bis 3 Oktalziffern
\xH		beliebig	H = ein String mit Hex-Ziffern
\XH		beliebig	H = ein String mit Hex-Ziffern

Spezielle Turbo C++ Konstanten mit zwei Zeichen

Turbo C++ unterstützt Konstanten mit zwei Zeichen (beispielsweise '*An*', '*\n\t*' und '*\007\007*'). Diese Konstanten repräsentieren 16-Bit-**int**-Werte, wobei das erste Zeichen im niederwertigen, das zweite Zeichen im höherwertigen Byte gespeichert wird.

Derartige Konstanten sind nicht portierbar.

Konstanten mit und ohne Vorzeichen

Konstanten, die aus einem einzigen Zeichen bestehen (beispielsweise '*A*', '*\t*' und '*\007*'), werden in C ebenfalls als **int** mit 16 Bit dargestellt. In diesem Fall gilt das niederwertige Byte als vorzeichenbehaftet, das heißt, wenn der Wert größer ist als 127 (Basis 10), wird das höherwertige Byte auf -1 (hexadezimal: 0xFF) gesetzt.

Diese Automatik kann ausgeschaltet werden, indem man als standardmäßigen *char*-Typ *unsigned char* wählt (Auswahl des Menüpunktes *Vorzeichenlose Zeichen* im Menü *Optionen/Compiler/Code-Generierung*). Das höherwertige Byte wird dann immer auf 0 gesetzt.

»Wide character«-Konstanten

Eine Zeichenkonstante, der ein *L* vorausgeht, ist eine »Wide character«-Konstante vom Datentyp *wchar_t* (der in stddef.h definiert ist). Ein Beispiel:

```
x = L 'A';
```

Gleitkomma-konstanten

Eine *Gleitkommakonstante* setzt sich aus fünf Bestandteilen zusammen:

- Vorkommastellen
- Dezimalpunkt
- Nachkommastellen
- *e* oder *E* und ein vorzeichenbehafteter Integerexponent (optional)
- Suffix: *f*, *F* oder *l*, *L* (optional)

Es können entweder die Vorkomma- oder Nachkommastellen wegfallen (aber nicht beide). Sie können entweder den Dezimalpunkt oder den Buchstaben *e* (*E*) und den Exponenten weglassen (aber nicht beide). Diese Regeln ermöglichen sowohl die konventionelle als auch die wissenschaftliche Notation (mit Exponenten). Negative Gleitkommakonstanten werden als positive Konstanten mit vorangestelltem unären Minusoperator behandelt. Dazu einige Beispiele:

Konstante	Wert
23.45e6	23.45×10^{6}
.0	0
0.	0
1.	$1.0 \times 10^{0} = 1.0$
-1.23	-1.23
2e-5	2.0×10^{-5}
3E+10	3.0×10^{10}
.09E34	0.09×10^{34}

Der Typ von Gleitkommakonstanten

Gleitkommakonstanten ohne Suffix sind vom Datentyp **double**. Durch Anfügen eines *f*- oder *F*-Suffixes können Sie einer Gleitkommakonstanten den Typ *float*, durch Anfügen eines *l*- oder *L*-Suffixes den Typ *long double* zuweisen. Im folgenden sehen Sie die Bereiche für **float, double** und **long double**.

Tabelle 1.8 Größe und Bereich der Turbo C++ Gleitkommakonstanten

Typ	Größe (Bits)	Bereich
float	32	$3.4 * 10^{-38}$ bis $3.4 * 10^{38}$
double	64	$1.7 * 10^{-308}$ bis $1.7 * 10^{308}$
long double	80	$3.4 * 10^{-4932}$ bis $1.1 * 10^{4932}$

Aufzählungstyp-Konstanten

Aufzählungstyp-Konstanten sind Bezeichner, die in **enum**-Deklarationen definiert werden. Diese Bezeichner werden normalerweise unter mnemotechnischen Aspekten gewählt. Die Aufzählungstyp-Konstanten sind Integerdatentypen. Sie können in allen Ausdrücken eingesetzt werden, in denen Integerkonstanten zulässig sind. Derselbe Bezeichner darf im Gültigkeitsbereich der **enum**-Deklaration nur einmal vorkommen. Negative Initialisierer sind erlaubt.

Im Abschnitt »Aufzählungstypen« in Kapitel 2 »Semantische Grammatik« finden Sie eine detaillierte Erläuterung der **enum**-Deklarationen.

Der von Aufzählungstyp-Konstanten übernommene Wert richtet sich nach der **enum**-Deklaration und dem Vorhandensein von (optionalen) *Initialisierern*. In dem Beispiel

```
enum team { giants, cubs, dodgers };
```

sind *giants*, *cubs* und *dodgers* Aufzählungstyp-Konstanten vom Typ *team*, die jeder Variablen vom Typ *team* oder jeder anderen Variablen eines Integertyps zugewiesen werden können.

Die Werte, die die Aufzählungstyp-Konstanten annehmen können, lauten ohne explizite Initialisierer:

```
giants = 0, cubs = 1, dodgers = 2
```

In dem Beispiel

```
enum team { giants, cubs=3, dodgers = giants + 1 };
```

sind die Konstanten folgendermaßen gesetzt:

```
giants = 0, cubs = 3, dodgers = 1
```

Die Konstantenwerte müssen nicht eindeutig sein:

```
enum team { giants, cubs = 1, dodgers = cubs - 1 };
```

String-konstanten

Stringkonstanten (oder *String-Literale*) sind eine spezielle Konstantenkategorie für feste Zeichenfolgen. Eine Stringkonstante ist vom Datentyp *array of char* und gehört zur Speicherklasse *static*. Derartige Konstanten setzen sich aus einer Reihe beliebig vieler Zeichen in Anführungszeichen zusammen:

```
"Dies ist eine Stringkonstante!"
```

Der Null-String (leerer String) ist: "". Die Zeichen, die innerhalb der Anführungszeichen stehen, können auch Escape-Sequenzen enthalten. Der Programmcode

```
"\t\t\"Name\"\\\tAdresse\n\n"
```

gibt aus:

```
"Name"\    Adresse
```

Vor *"Name"* stehen zwei Tabulatoren, vor *Adresse* ein Tabulator. Danach folgen zwei neue Zeilen. \" sorgt für die Ausgabe der Anführungszeichen im String. Stringkonstanten werden intern als die gegebene Zeichensequenz plus einem abschließenden Null-Zeichen (*\0*) abgespeichert. Ein leerer String wird als ein *\0*-Zeichen abgelegt.

Stringkonstanten, die nur durch Whitespace getrennt sind, werden während des Parsens verkettet. Das Programm

```
#include <stdio.h>

main()
{
   char     *p;
   p = "Dies ist ein Beispiel dafür, wie Turbo C++"
       " automatisch die\nVerkettung sehr langer"
       " Strings durchführt. Dadurch wird Ihr Pro-\ngramm"
       " übersichtlicher gestaltet.";
   printf(p);
   return(0);
}
```

liefert folgenden Ausdruck:

```
Dies ist ein Beispiel dafür, wie Turbo C++ automatisch die
Verkettung sehr langer Strings durchführt. Dadurch wird Ihr Pro-
gramm übersichtlicher gestaltet.
```

Durch Verwendung des Backslash (\) können Sie eine Stringkonstante auf zwei Zeilen aufteilen:

```
puts("Dies ist in Wirklichkeit \
ein einzeiliger String");
```

Konstanten und ihre interne Darstellung

ANSI C »weiß«, daß die Größe und der numerische Bereich der grundlegenden Datentypen (und ihrer verschiedenen Varianten) implementierungsspezifisch sind und sich normalerweise von der Architektur des Rechners ableiten. Für Turbo C++ ist die IBM-PC-Familie (und alle kompatiblen Rechner) der Ausgangspunkt. Somit bestimmt die Architektur der Intel 8088 und 80x86 Mikroprozessoren die Auswahl der internen Darstellung für die verschiedenen Datentypen.

In der folgenden Tabelle sehen Sie die Größen und die sich daraus ergebenden Bereiche für die grundlegenden Datentypen in Turbo C++. Im Abschnitt »Die grundlegenden Typen« in Kapitel 2 »Semantische Grammatik« werden diese Datentypen ausführlich erläutert.

Abbildung 1.1 zeigt die interne Darstellung der Typen.

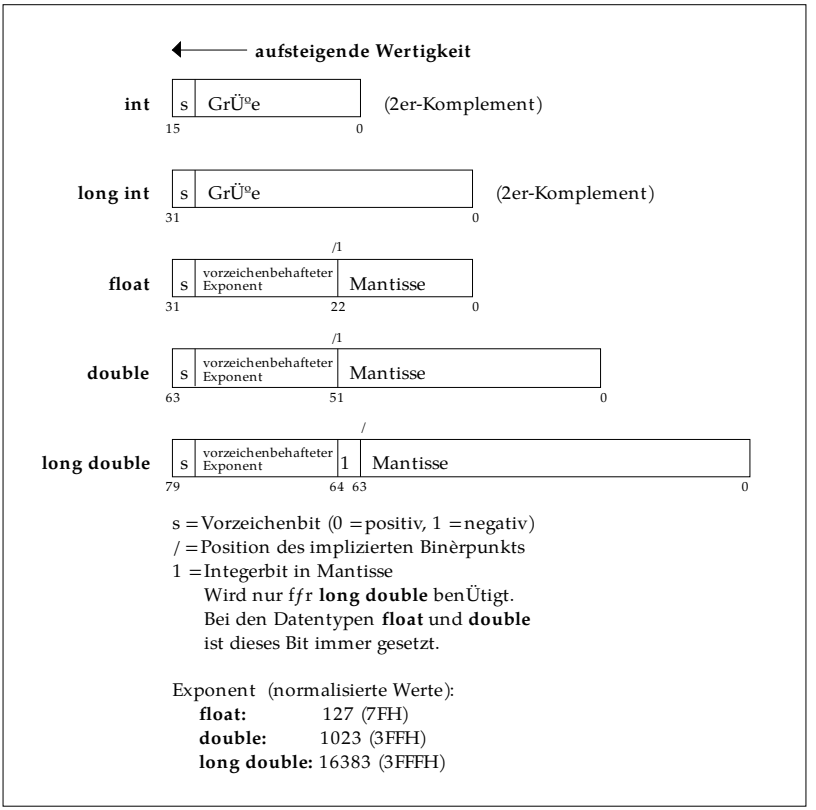


Abbildung 1.1 Interne Darstellung der Datentypen

Tabelle 1.9 Datentypen, Größen und Bereiche

Typ	Bits	Bereich	Beispielanwendungen
unsigned char	8	0 bis 255	Kleine Zahlen und kompletter PC-Zeichensatz
char	8	-128 bis 127	Sehr kleine Zahlen und ASCII-Zeichen
enum	16	-32768 bis 32767	Geordnete Wertemengen
unsigned int	16	0 bis 65535	Große Zahlen und Schleifen
sort int	16	-32768 bis 32767	Zähler, kleine Zahlen, Schleifensteuerung
int	16	-32768 bis 32767	Zähler, kleine Zahlen, Schleifensteuerung
unsigned long	32	0 bis 4294967295	Astronomische Distanzen

Typ	Bits	Bereich	Beispielanwendungen
long	32	-2147483648 bis 2147483647	Sehr große Zahlen, statistische Grundgesamtheiten
float	32	3.4×10^{-38} bis 3.4×10^{38}	Wissenschaftlich, 7-stellige Genauigkeit
double	64	1.7×10^{-308} bis 1.7×10^{308}	Wissenschaftlich, 15-stellige Genauigkeit
long double	80	3.4×10^{-4932} bis 1.1×10^{4932}	Finanzrechnung, 19-stellige Genauigkeit
near pointer	16	Nicht anwendbar	Speicheradressen
far pointer	32	Nicht anwendbar	Außerhalb des aktuellen Segments

Konstantenausdrücke

Ein *Konstantenausdruck* ist ein Ausdruck, der zu einer Konstante ausgewertet wird. Er muß einen Wert ergeben, der im Wertebereich seines Typs liegt. Konstantenausdrücke werden genauso ausgewertet wie reguläre Ausdrücke. Konstantenausdrücke können überall dort verwendet werden, wo eine Konstante legal ist. Die Syntax für einen Konstantenausdruck lautet:

Konstantenausdruck:
Bedingter Ausdruck

Konstantenausdrücke dürfen keinen der folgenden Operatoren enthalten, es sei denn, daß die Operatoren Operanden des *sizeof*-Operators sind:

- Zuweisungsoperator
- Komma-Operator
- Dekrement-Operator
- Funktionsaufruf
- Inkrement-Operator

Interpunktionszeichen

Interpunktionszeichen (auch Trennzeichen genannt) sind in Turbo C++ wie folgt definiert:

Interpunktionszeichen: eines der folgenden Symbole
`[] () {} , ; : ... * = #`

Eckige Klammern

`[ ]` beinhalten einfache und mehrdimensionale Array-Indizes:

```
char ch, str[] = "Stan"
int mat[3][4];      /* 3x4-Matrix */
ch = str[3];        /* 4. Element */
```

Runde Klammern
( ) fassen Ausdrücke zusammen, isolieren konditionale Ausdrücke und repräsentieren Funktionsaufrufe und Funktionsparameter:

```
d = c * (a + b);     /* ändern der normalen Reihenfolge */

if (d == z) ++x;     /* Erforderlich bei einer
                        bedingten Anweisung */

func();              /* Funktionsaufruf ohne Argumente */
int (*fptr)();       /* Funktionszeiger-Deklaration */
fptr = func;         /* keine () bedeutet Funktionszeiger */

void func2 (int n); /* Funktionsdeklaration mit Argumenten */
```

Runde Klammern müssen in Makrodefinitionen verwendet werden, damit potentielle Rangfolgeprobleme bei der Erweiterung ausgeschlossen sind:

```
#define WUERFEL(x)  ( (x) * (x) * (x) )
```

Der Einsatz von runden Klammern zur Änderung der normalen Operator-Rangfolge und der Assoziativitätsregeln werden im Abschnitt »Ausdrücke« in Kapitel 2 »Semantische Grammatik« erläutert.

Geschweifte Klammern
{ } markieren den Beginn und das Ende eines Anweisungsblocks:

```
if (d == z)
{
   ++X;
   func();
}
```

Die schließende Klammer dient als Endekriterium (Terminator) für den Anweisungsblock; aus diesem Grund muß nach } kein Strichpunkt (;) mehr stehen, außer bei Struktur- oder Klassendeklarationen. Häufig ist ein Strichpunkt unzulässig, wie z.B.

```
if (Ausdruck)
  {};                /* Strichpunkt nicht zulässig */
else
```

Komma
Das Komma (,) trennt die Elemente in einer Funktions-Argumentliste:

```
void func(int n, float f, char ch);
```

Das Komma wird auch als Operator in *Komma-Ausdrücken* verwendet. Die Kombination beider Komma-Anwendungen ist erlaubt, wenn gleichzeitig runde Klammern zur Unterscheidung eingesetzt werden:

```
func(i, j);          /* Aufruf einer Funktion mit zwei Argumenten */
func((exp1, exp2), (exp3, exp4, exp5));
                     /* ebenfalls Aufrufe mit zwei Argumenten */
```

Strichpunkt

Der Strichpunkt (;) dient als Endekriterium einer Anweisung. Jeder gültige C- oder C++-Ausdruck (auch der leere Ausdruck), an dessen Ende ein Strichpunkt steht, wird als Anweisung (*Ausdruck-Anweisung*) interpretiert. Der Ausdruck wird nur ausgewertet und sein Wert wird verworfen. Wenn der Ausdruck keine Nebeneffekte hat, wird er möglicherweise von Turbo C++ ignoriert.

```
a + b;          /* a + b wird ausgewertet, aber der Wert
                   wird verworfen */
++a;            /* Nebeneffekt auf a, aber der Wert von
                   ++a wird verworfen */
;               /* leerer Ausdruck = leere Anweisung */
```

Strichpunkte werden oft zur Erstellung einer leeren Anweisung verwendet:

```
for (i=0; i < n; i++)
{
   ;
}
```

Doppelpunkt

Mit dem Doppelpunkt (:) wird ein Label gekennzeichnet.

```
start:     x=0;
...
goto start;
...
switch (a) {
   case 1: puts ("Eins");
           break;
   case 2: puts ("Zwei");
           break;
...
default:   puts ("Nichts davon!");
           break;
}
```

Labels werden im Abschnitt »Anweisungs-Label« in Kapitel 2 »Semantische Grammatik« erläutert.

Ellipsen

Ellipsen (...) sind drei aufeinanderfolgende Punkte ohne Whitespace. Ellipsen kennzeichnen in den formalen Argumentlisten der Funktionsprototypen eine variable Anzahl von Argumenten oder Argumente mit nicht festgelegten Typen. Die Deklaration

```
void func(int n, char ch,...);
```

besagt, daß Aufrufe von *func* mindestens zwei Argumente haben müssen, ein **int** und ein **char**. Daneben können beliebig viele Argumente vorhanden sein.

Hinweis: In C++ ist das Komma vor den Ellipsen optional.

Sternchen (Zeigerdeklaration)
Das Sternchen (*) in einer Variablendeklaration zeigt an, daß ein Zeiger auf einen Typ generiert wird:

```
*char_ptr;              /* es wird ein Zeiger auf char deklariert */
```

Zeiger mit mehrfachen Indirektionsebenen werden durch mehrere Sternchen gekennzeichnet:

```
int **int_ptr;          /* ein Zeiger auf einen Zeiger auf ein int */
double ***double_ptr;   /* ein Zeiger auf einen Zeiger auf
                           auf einen Zeiger auf ein double */
```

Ein Sternchen kann auch als Multiplikationsoperator oder als Operator dienen, der einen Zeiger dereferenziert:

```
i = *int_ptr;
a = b * 3.14;
```

Gleichheitszeichen
Das Gleichheitszeichen (=) trennt Variablendeklarationen von Initialisierungslisten:

```
char array[5] = { 1, 2, 3, 4, 5 };
int x = 5;
```

In C++ können – mit wenigen Einschränkungen – alle Deklarationstypen an jeder beliebigen Stelle im Programm stehen. In einer C-Funktion darf vor den Variablendeklarationen kein Programmcode stehen.
In der Argumentliste einer C++ Funktion kennzeichnet das Gleichheitszeichen den Vorgabewert für einen Parameter:

```
int f(int i = 0) {...}   /* Der Parameter i hat den
                            Standardwert 0 */
```

Das Gleichheitszeichen wird auch als Zuweisungsoperator in Ausdrücken verwendet:

```
a = b + c;
ptr = farmalloc(sizeof(float)*100);
```

#-Zeichen (Präprozessor-Direktive)
Das #-Zeichen fungiert als Präprozessor-Direktive, wenn es als erstes Nicht-Whitespace-Zeichen in einer Zeile steht. Es gibt eine Compileraktion an, die nicht notwendigerweise mit der Codegenerierung verknüpft ist.
Informationen dazu finden Sie in Kapitel 4 »Der Präprozessor«. Die Zeichen # und ## werden auch als Operatoren eingesetzt, die den Präprozessor anweisen, Symbole zu ersetzen bzw. zu verschmelzen.

2. Semantische Grammatik

In diesem Kapitel wird die formale Definition der Turbo C++Sprachstruktur erläutert. Diese semantische Grammatik bestimmt die Regeln, nach denen Symbole zur Bildung von Ausdrücken, Anweisungen oder anderen bedeutungstragenden Einheiten kombiniert werden können. Im Gegensatz dazu befaßt sich die lexikalische Grammatik (erläutert in Kapitel 1 »Lexikalische Grammatik« im vorliegenden Handbuch) mit den verschiedenen Kategorien der wortähnlichen Einheiten, den *Tokens* oder *Symbolen*, aus denen die Sprache besteht.

Deklarationen

Im folgenden Abschnitt finden Sie eine kurze Zusammenfassung über Themen, die mit Deklarationen zu tun haben: Objekte, Typen, Speicherklassen, Gültigkeitsbereiche, Sichtbarkeit, Dauer und Bindung. Mit diesen Begriffen sollten Sie prinzipiell vertraut sein, bevor Sie sich mit der Deklarationssyntax beschäftigen.

Gültigkeitsbereich, Sichtbarkeit, Dauer und Bindung legen diejenigen Programmteile fest, von denen aus ein zulässiger Zugriff auf das mit dem Bezeichner verknüpfte Objekt möglich ist.

Objekte

Ein *Objekt* ist ein identifizierbarer Speicherbereich, in dem ein fester oder variabler Wert (oder eine Menge von Werten) steht. (Die Bezeichnung *Objekt* an dieser Stelle darf nicht mit dem generelleren Begriff »Objekt« – wie er in objektorientierten Sprachen verwendet wird – verwechselt werden.) Jedes Objekt hat einen Namen und einen Typ (den sogenannten *Datentyp*). Der Zugriff auf das Objekt erfolgt über seinen Namen. Dieser Name kann ein einfacher Bezeichner oder ein komplexer Ausdruck sein, der »eindeutig« auf das Objekt zeigt. Der Typ wird dazu verwendet, um

- die korrekte Speicherreservierung festzulegen, die zu Beginn erforderlich ist;
- die Bitmuster zu interpretieren, die bei späteren Zugriffen im Objekt gefunden werden;

- durch Überprüfung der Typen sicherzustellen, daß zulässige Zuweisungen erfolgen.

Turbo C++ unterstützt sowohl vordefinierte als auch benutzerdefinierte Typen. Dazu gehören Integerzahlen mit und ohne Vorzeichen in verschiedenen Größen, Gleitkommazahlen von unterschiedlicher Genauigkeit, Strukturen, Varianten, Arrays und Klassen. Zusätzlich können Zeiger auf fast alle Objekte definiert und in unterschiedlichen Speichermodellen verwendet werden.

Die Standardbibliotheken von Turbo C++sowie Ihre eigenen Programm- und Header-Dateien müssen eindeutige Bezeichner (oder von ihnen abgeleitete Ausdrücke) und Typen enthalten. Nur dann ist der konsistente Zugriff, die Interpretation und (gegebenenfalls) die Änderung des einem aktiven Objekt Ihres Programms entsprechenden Bitmusters im Speicher sichergestellt.

Deklarationen stellen die Verbindung zwischen Bezeichnern und Objekten her. Jede Deklaration verknüpft einen Bezeichner mit einem Datentyp. Darüber hinaus bestimmen die meisten Deklarationen, die sogenannten *Definitionsdeklarationen*, auch die Generierung des Objekts (wo und wann), das heißt, sie übernehmen die Zuweisung des Speicherplatzes und die mögliche Initialisierung. Andere Deklarationstypen, sogenannte *Referenzdeklarationen*, geben dem Compiler lediglich die Bezeichner und Typen bekannt. Für einen Bezeichner kann es viele Referenzdeklarationen geben, besonders in einem Programm mit vielen Dateien. Es darf aber nur eine Definitionsdeklaration für diesen Bezeichner geben.

Generell gilt: Ein Bezeichner kann nicht vor seiner Deklaration verwendet werden. Zulässige Ausnahmen von dieser Regel, sogenannte *forward*-Referenzen, sind Labels, Aufrufe von nicht deklarierten Funktionen und die »Tags« von, Klassen, Strukturen und Varianten.

L-Werte

Ein *L-Wert* (lvalue) ist ein Ausdruck, der ein Objekt bezeichnet. Ein Beispiel für einen L-Wert-Ausdruck ist **P*, wobei *P* ein Ausdruck für einen Zeiger ungleich Null ist. Ein *modifizierbarer L-Wert* ist ein Bezeichner oder ein Ausdruck, der zu einem Objekt gehört, auf das zugegriffen werden kann und das im Speicher geändert werden darf. Ein **const**-Zeiger auf eine Konstante ist zum Beispiel *kein* modifizierbarer L-Wert. Ein Zeiger auf eine Konstante kann geändert werden, nicht aber deren dereferenzierter Wert.

Historisch steht *L* für »links«, das heißt, daß ein L-Wert auf der linken Seite einer Zuweisungsanweisung stehen darf. Jetzt dürfen nur noch modifizierbare L-Werte auf der linken Seite einer Zuweisungsanweisung stehen. Sind *a* und *b* beispielsweise Integer-Bezeichner (und keine Konstanten) mit entsprechend zugewiesenem Speicherplatz, dann handelt es sich bei ihnen um modifizierbare L-Werte. Somit sind Zuweisungen wie etwa *a =1;* und *b =a +b;* zulässig.

R-Werte

Der Ausdruck *a + b* ist kein L-Wert. *a + b = a* ist nicht zulässig, weil der Ausdruck auf der linken Seite mit keinem Objekt verbunden ist. Derartige Ausdrücke werden oft als *R-Werte* bezeichnet (als Abkürzung für »rechte Werte«).

Typen und Speicherklassen

Mit Objekten verknüpfte Bezeichner müssen zwei Attribute haben: *Speicherklasse* und *Typ* (Datentyp). Der Turbo C++ Compiler holt sich diese Attribute aus den impliziten oder expliziten Deklarationen im Quellcode.

Die Speicherklasse legt die Position (Datensegment, Register, Heap oder Stack) und die Dauer oder die »Lebensdauer« des Objekts fest (das kann die gesamte Laufzeit des Programms oder die Ausführungszeit eines Blocks sein). Die Speicherklasse kann entweder durch die Syntax der Deklaration oder durch deren Position im Quellcode angegeben werden, bzw. auf beide Arten. Der Typ bestimmt, wieviel Speicherplatz einem Objekt zugeordnet wird und wie das Programm die im Speicherbereich des Objekts vorhandenen Bitmuster interpretieren soll. Ein Datentyp kann als die Menge der Werte angesehen werden (sie hängt häufig von der Implementierung ab), die Bezeichner dieses Typs annehmen können und als die Menge der Operationen, die mit diesen Werten durchgeführt werden dürfen. Mit dem speziellen Operator **sizeof** (der während der Compilierung ausgeweitet wird) können Sie für jeden benutzerdefinierten oder standardmäßigen Typ die Größe in Byte ermitteln; weitere Informationen finden Sie im Abschnitt »Der **sizeof**-Operator« weiter hinten in diesem Kapitel.

Gültigkeitsbereich

Der *Gültigkeitsbereich* eines Bezeichners ist der Programmteil, in dem über den Bezeichner auf das Objekt zugegriffen werden kann. Es gibt fünf Kategorien für den Gültigkeitsbereich: *Block* (oder *lokal*), *Funktion*, *Funktionsprototyp*, *Datei* und *Klasse* (nur in C++). In welcher Kategorie der Gültigkeitsbereich fällt, hängt davon ab, wie und wo Bezeichner deklariert werden.

Gültigkeitsbereich: Block

Der Block-Gültigkeitsbereich beginnt beim Deklarationspunkt und hört am Ende des Blocks auf, der die Deklaration enthält (ein derartiger Block wird *einschließender* Block genannt). Hierzu gehören auch Parameterdeklarationen innerhalb einer Funktionsdefinition.

Gültigkeitsbereich: Funktion

Labels für Anweisungen sind die einzigen Bezeichner, die eine Funktion als Gültigkeitsbereich haben. Labelnamen können mit **goto**-Anweisungen an beliebiger Stelle in der Funktion, in der das Label deklariert wird, verwendet werden. Die Deklaration von Labels erfolgt implizit durch die Angabe von *label_name:*, gefolgt von einer Anweisung. Derselbe Labelname darf in einer Funktion nur einmal vorkommen.

Gültigkeitsbereich: Funktionsprototyp

Bezeichner, die in der Liste der Parameterdeklarationen eines Funktionsprototyps (die nicht Teil einer Funktionsdefinition ist) deklariert werden, haben als

Gültigkeitsbereich den Funktionsprototyp. Dieser Gültigkeitsbereich endet mit dem Ende des Funktionsprototyps.

Gültigkeitsbereich: Datei
Bezeichner mit dem Gültigkeitsbereich Datei, sogenannte *globale* Bezeichner, werden außerhalb aller Blöcke und Klassen deklariert. Ihr Gültigkeitsbereich erstreckt sich vom Deklarationspunkt bis zum Ende der Quelldatei.

Gültigkeitsbereich: Klasse (nur C++)
Für diesen Zweck definieren wir eine Klasse als eine benannte Ansammlung von Elementen, einschließlich Datenstrukturen und Funktionen, die auf diese zugreifen. Der Gültigkeitsbereich Klasse wird – mit einigen Ausnahmen – auf die Namen von Elementen einer bestimmten Klasse angewendet. Für Klassen und ihre Objekte gelten viele spezielle Zugriffs- und Gültigkeitsbereichsregeln. Lesen Sie dazu den Abschnitt »Klassen« in Kapitel 3 »C++«.

Gültigkeits- und Namensbereich
Der *Namensbereich* ist der Bereich, in dem derselbe Bezeichner nur einmal vorkommen darf. Es gibt in C vier verschiedene Bezeichnerklassen:

1. **goto**-Labelnamen. Diese Bezeichner müssen innerhalb der Funktion, in der sie deklariert werden, eindeutig sein.

2. Struktur-, Varianten- und Aufzählungs-»Tags«. Diese Bezeichner müssen innerhalb des Blocks, in dem sie definiert werden, eindeutig sein. »Tags«, die außerhalb einer Funktion deklariert werden, müssen innerhalb aller extern definierten »Tags« eindeutig sein. Strukturen, Klassen und Aufzählungstypen stehen in C++ in demselben Namensbereich.

3. Namen von Struktur- und Variantenelementen. Diese Bezeichner müssen innerhalb der Struktur oder Variante, in der sie definiert werden, eindeutig sein. Für die Namen von Elementen in verschiedenen Strukturen gibt es keine Restriktionen.

4. Variablen, **typedef**-Namen und Aufzählungselemente. Diese Bezeichner müssen innerhalb des Gültigkeitsbereichs, in dem sie definiert werden, eindeutig sein. Extern deklarierte Bezeichner müssen innerhalb der extern deklarierten Variablen eindeutig sein.

Sichtbarkeit

Unter der *Sichtbarkeit* eines Bezeichners versteht man den Bereich des Quellcodes, von dem aus ein zulässiger Zugriff auf das mit dem Bezeichner verknüpfte Objekt möglich ist.

Gültigkeitsbereich und Sichtbarkeit fallen normalerweise zusammen. Es gibt jedoch Konstellationen, bei denen ein Objekt vorübergehend durch das Auftreten eines doppelten Bezeichners *verborgen* wird. Das Objekt existiert weiterhin, aber es kann kein Zugriff mit dem ursprünglichen Bezeichner erfolgen. Dieser ist erst möglich, wenn der Gültigkeitsbereich des doppelten Bezeichners endet.

Die Sichtbarkeit kann nicht größer sein als der Gültigkeitsbereich, während dies umgekehrt möglich ist.

```
...
{
   int i; char ch;     // per Voreinstellung automatisch
   i = 3;              // int i und char ch im
                       // Gültigkeitsbereich und sichtbar
...
    {
     double i;
     i = 3.0e3;        // double i innerhalb des
                       // Gültigkeitsbereichs und sichtbar
                       // int i = 3 im Gültigkeitsbereich,
                       // aber verborgen
     ch = 'A';         // char ch im Gültigkeitsbereich und
                       // und sichtbar
    }
                       // double i außerhalb des
                       // Gültigkeitsbereichs
    i += 1;            // int i sichtbar und = 4
...                    // char ch noch im Gültigkeits-
                       // bereich und sichtbar = 'A'
}
...                    // int i und char ch außerhalb
                       // des Gültigkeitsbereichs
```

Hinweis: In C++ gelten für verborgene Klassen- und Klassenelementnamen besondere Regeln: Spezielle C++ Operatoren ermöglichen es, daß unter bestimmten Umständen auf verborgene Bezeichner zugegriffen werden kann. Näheres finden Sie im Abschnitt »Klassen« in Kapitel 3 »C++«.

Dauer

Die *Dauer* ist eng mit der Speicherklasse verwandt und definiert den Zeitraum, in dem deklarierten Bezeichnern im Speicher reale, physikalische Objekte zugewiesen sind. Es wird zwischen Compilierungszeit- und Laufzeitobjekten unterschieden. Beispielsweise wird Variablen – im Gegensatz zu **typedef** und Typen – während der Laufzeit Speicherplatz zugewiesen. Es gibt drei Arten von Dauer: *statisch, lokal* und *dynamisch.*

Statische Dauer

Objekten mit *statischer* Dauer wird Speicherplatz zugewiesen, sobald die Ausführung beginnt; die Speicherplatzzuweisung bleibt bestehen, bis das Programm endet. Sie werden in festen Datensegmenten gespeichert, denen entsprechend dem benutzten Speichermodell Platz zugewiesen wurde.

Alle Funktionen – unabhängig davon, wo sie definiert werden – sind Objekte mit statischer Dauer. Variablen mit dem Gültigkeitsbereich Datei haben eben-

falls statische Dauer. Anderen Variablen kann durch die Zuweisung der Speicherklasse **static** oder **extern** statische Dauer zugeordnet werden. Objekte von statischer Dauer werden auf Null gesetzt, wenn kein expliziter Initialisierer oder Konstruktor (in C++) vorhanden ist.

Statische Dauer darf nicht mit dem Datei- oder globalen Gültigkeitsbereich verwechselt werden. Ein Objekt von statischer Dauer kann gleichzeitig einen lokalen Gültigkeitsbereich haben.

Lokale Dauer

Objekte mit *lokaler* Dauer, auch als *automatische* Objekte bezeichnet, werden auf dem Stack (oder in einem Register) angelegt, sobald das Programm in einen *einschließenden* Block oder in eine Funktion eintritt. Diese Objekte verlieren ihren Speicherplatz, wenn das Programm den Block oder die Funktion verläßt. Objekte mit lokaler Dauer müssen explizit initialisiert werden, da ansonsten ihr Inhalt nicht vorhersehbar ist. Derartige Objekte müssen einen lokalen Gültigkeitsbereich bzw. einen Funktions-Gültigkeitsbereich haben. Der Speicherklassen-Spezifizierer **auto** kann bei der Deklaration von Variablen mit lokaler Dauer verwendet werden. Seine Verwendung ist aber normalerweise überflüssig, da **auto** für Variablen, die innerhalb von Blöcken deklariert sind, voreingestellt ist.

Ein Objekt von lokaler Dauer hat einen lokalen Gültigkeitsbereich, weil es nicht außerhalb des einschließenden Blocks existiert. Die Umkehrung gilt jedoch nicht: Ein Objekt mit lokalem Gültigkeitsbereich kann eine statische Dauer haben.

Bei der Deklaration von Variablen (zum Beispiel **int**, **char**, **float**) impliziert der Speicherklassen-Spezifizierer **register** gleichzeitig **auto**; der Compiler wird jedoch aufgefordert, dem Objekt – wenn möglich – ein Register zuzuweisen. Turbo C++ kann so eingestellt werden, daß einer lokalen Integer- oder Zeigervariablen immer ein Register zugewiesen wird, wenn dies möglich ist. Ist kein Register frei, wird die Variable ohne Warnung oder Fehlermeldung als lokales Objekt der Speicherklasse **auto** zugeordnet.

Dynamische Dauer

Objekte mit *dynamischer* Dauer werden während eines Programms mit besonderen Funktionsaufrufen angelegt und wieder freigegeben. Die Speicherzuweisung erfolgt entweder mit Standardbibliotheksfunktionen wie **malloc** oder mit dem C++Operator **new**. Der zugewiesene Speicherplatz liegt in einem bestimmten Speicherbereich, dem sogenannten *Heap*. Mit **free** oder **delete** wird der Speicherplatz wieder freigegeben.

Übersetzungseinheiten

Der Ausdruck *Übersetzungseinheit* bezieht sich auf eine Quelltextdatei und deren Include-Dateien. Nicht dazu zählen Quelltextzeilen, die von bedingten Präprozessor-Direktiven ausgeschlossen werden.

Syntaktisch wird eine Übersetzungseinheit als eine Abfolge externer Deklarationen definiert:

Übersetzungseinheit:
Extern-Deklaration
Übersetzungseinheit Extern-Deklaration

Extern-Deklaration
Funktionsdefinition
Deklaration

Das Wort *extern* hat in C verschiedene Bedeutungen. Im vorliegenden Kontext wird es für Deklarationen verwendet, die außerhalb einer Funktion definiert werden und die deshalb den Gültigkeitsbereich Datei haben. (Beachten Sie, daß *extern* in Zusammenhang mit der Bindung eine besondere Bedeutung hat; näheres dazu erfahren Sie im nächsten Abschnitt.) Jede Deklaration, die Speicher für ein Objekt oder eine Funktion bereitstellt, wird als Definition (oder definierende Deklaration) bezeichnet. Weitere Informationen finden Sie im Abschnitt »Externe Deklarationen und Definitionen«.

Bindung

Ein ausführbares Programm entsteht normalerweise durch die Compilierung von verschiedenen, voneinander unabhängigen Übersetzungseinheiten und anschließendes Linken der resultierenden Objektdateien mit bereits vorhandenen Bibliotheken. Probleme können entstehen, wenn derselbe Bezeichner in verschiedenen Gültigkeitsbereichen (etwa in unterschiedlichen Dateien) oder mehr als einmal im selben Gültigkeitsbereich deklariert ist. *Bindung* ist der Vorgang, der die korrekte Verknüpfung zwischen jedem Auftreten eines Bezeichners und einem bestimmten Objekt oder einer bestimmten Funktion herstellt. Jeder Bezeichner hat eines der drei Bindungsattribute, die in engem Zusammenhang mit dem Gültigkeitsbereich stehen: *externe*, *interne* oder *keine* Bindung. Diese Attribute werden von der Position und dem Format Ihrer Deklarationen, sowie dem expliziten (oder durch Voreinstellung impliziten) Einsatz der Speicherklassen-Spezifizierer **static** oder **extern** bestimmt.

Jedes Auftreten eines bestimmten Bezeichners mit externer Bindung repräsentiert in allen Dateien und Bibliotheken eines Programms dasselbe Objekt oder dieselbe Funktion. Jedes Auftreten eines Bezeichners mit interner Bindung repräsentiert dasselbe Objekt oder dieselbe Funktion nur in einer Datei. Bezeichner mit keiner Bindung repräsentieren eindeutige Einheiten. Es gelten folgende Regeln für externe und interne Bindung:

1. Alle Objekt- oder Dateibezeichner, die den Gültigkeitsbereich Datei haben, verfügen über interne Bindung, wenn ihre Deklaration den Speicherklassen-Spezifizierer **static** enthält. Kommt derselbe Bezeichner innerhalb derselben Datei mit interner und externer Bindung vor, so erhält er in C interne Bindung und in C++ externe Bindung.

2. Enthält die Deklaration eines Objekt- oder Funktionsbezeichners den Speicherklassen-Spezifizierer **extern**, hat der Bezeichner dieselbe Bindung wie irgendeine sichtbare Deklaration des Bezeichners mit dem Gültigkeitsbereich Datei. Gibt es keine derartige Deklaration, hat der Bezeichner externe Bindung.

3. Wird eine Funktion ohne Speicherklassen-Spezifizierer deklariert, wird ihre Bindung so festgelegt, als ob der Spezifizierer **extern** verwendet wurde.

4. Wenn ein Objektbezeichner mit dem Gültigkeitsbereich Datei ohne einen Speicherklassen-Spezifizierer deklariert wird, hat er externe Bindung.

Folgende Bezeichner haben das Attribut *keine Bindung*:

1. Jeder Bezeichner, der nicht als Objekt- oder Funktionsbezeichner deklariert wird (zum Beispiel ein **typedef**-Bezeichner);

2. Funktionsparameter;

3. Bezeichner mit dem Gültigkeitsbereich Block für Objekte, die nicht zu der Speicherklasse **extern** gehören.

Ergänzen von Namen

Wenn ein C++Modul compiliert ist, erzeugt der Compiler Funktionsnamen, die die Typen der Funktionsargumente in verschlüsselter Form enthalten. Dieser Vorgang wird als Namensergänzung bezeichnet (im englischen Sprachraum ist dafür die Bezeichnung »name mangling« gebräuchlich).

Notwendig geworden ist die Namensergänzung in C++ vor allen Dingen wegen der überladenen Funktionen. Eine Funktion zu überladen bedeutet, mehrere Funktionen, die eine ähnliche Aufgabe ausführen, unter einem gemeinsamen Namen anzusprechen. Dadurch wird die Lesbarkeit von Programmen – für menschliche Leser – erleichtert. Der Linker benötigt jedoch eindeutige Namen, und um diese zu erzeugen, verschlüsselt der Compiler die Parameterliste und den Rückgabewert in den ursprünglichen Namen hinein. Gleichzeitig kann der Linker falsche Funktionsaufrufe leichter feststellen.

Es gibt andererseits aber Situationen, in denen die Namensergänzung nicht wünschenswert ist. Wenn ein C++Modul mit einem Modul gelinkt werden soll, das keine ergänzten Namen enthält, muß dem Compiler mitgeteilt werden, daß er die Funktionsnamen aus den anderen Modulen nicht ergänzen soll. Eine derartige Situation ist dann gegeben, wenn Bibliotheken oder .OBJ-Dateien gelinkt werden, die mit einem C-Compiler compiliert wurden.

Wenn der Compiler den Namen einer Funktion nicht ergänzen soll, deklarieren Sie die Funktion als **extern** "C", z.B. folgendermaßen:

```
extern "C" void Cfunc( int );
```

Diese Deklaration veranlaßt den Compiler dazu, daß Referenzen auf die Funktion *Cfunc* nicht ergänzt werden sollen. Sie können die Deklaration **extern** "C" auch auf einen Namensblock anwenden:

```
extern "C" {
   void Cfunc1( int);
   void Cfunc2( int);
   void Cfunc3( int);
};
```

Wie bei der Deklaration für eine einzige Funktion wird auch hier dem Compiler mitgeteilt, daß Referenzen auf die Funktionen *Cfunc1*, *Cfunc2* und *Cfunc3* nicht ergänzt werden sollen. Diese Art der Blockdeklaration können Sie auch verwenden, wenn der Funktionsnamenblock in einer Header-Datei steht:

```
extern "C" {
   #include "locallib.h"
};
```

Deklarations-syntax

Die sechs miteinander verknüpften Attribute (Speicherklasse, Typ, Gültigkeitsbereich, Sichtbarkeit, Dauer und Bindung) werden – in verschiedener Art und Weise – über *Deklarationen* festgelegt.

Es gibt zwei Gruppen von Deklarationen: *definierende Deklarationen* (oder einfacher: *Definitionen*) und *Referenzdeklarationen* (manchmal werden sie auch *nicht-definierende Deklarationen* genannt). Eine definierende Deklaration beinhaltet – wie der Name bereits besagt – sowohl eine Deklaration als auch eine Definition; Referenzdeklarationen (oder nicht-definierende Deklarationen) benötigen an einer anderen Stelle im Programm eine Definition. Deklarationen führen Bezeichnernamen in das Programm ein und ordnen diese einem Typ zu. Eine Definition bewirkt darüber hinaus eine Speicherplatzzuweisung, d.h. sie erzeugt ein vollständiges Objekt mit Namen, Typ und Speicherplatz.

Tentative Definitionen

Der ANSI-C-Standard führt einen neuen Begriff ein: die *tentative Definition*. Eine externe Datendeklaration, die keinen Speicherklassen-Spezifizierer und Initialisierer hat, wird als tentative Definiton betrachtet. Wenn der deklarierte Bezeichner später definiert wird, so wird die tentative Definition behandelt, als wäre der Speicherklassen-Spezifizierer **extern** angegeben worden. Das heißt: die tentative Definition wird zu einer einfachen Referenzdeklaration.

Wird das Ende der Übersetzungseinheit erreicht und keine Definition mit einem Initialisierer für den Bezeichner gefunden, so wird die tentative Definition zu einer vollen Definition. Der Speicher für das Objekt wird reserviert, aber nicht initialisiert (d.h. mit Null gefüllt). Betrachten Sie folgendes Beispiel:

```
int x;
int x;      /* legal, eine Kopie von x wird reserviert */

int y;
int y = 4;  /* legal, y wird mit 4 initialisiert */

int z = 5;
int z = 6;  /* illegal, z wird zum 2. Mal initialisiert */
```

Hinweis: Im Gegensatz zu ANSI C verfügt C++ nicht über tentative Deklarationen; eine externe Datendeklaration ohne Speicherklassenbezeichner ist immer eine Definition.

Mögliche Deklarationen

Folgende Objekte können deklariert werden:

- Variable
- Funktionen
- Klassen und Klassenelemente (C++)
- Typen
- Struktur-, Varianten- und Aufzählungs-»Tags« (Identifizierungskennzeichen)
- Strukturelemente
- Variantenelemente
- Arrays anderer Typen
- Aufzählungstyp-Konstanten
- Labels
- Präprozessor-Makros

In den folgenden Tabellen finden Sie die vollständige Syntax der Deklarationen. Die rekursive Natur der Syntax erlaubt komplexe Deklaratoren. Sie sollten deshalb **typedef** verwenden, um die Lesbarkeit zu verbessern.

Tabelle 2.1 Deklarationssyntax von Turbo C++

Deklaration:
<Mehrere-Deklarations-Spezifizierer><Deklaratorliste>;
asm-*Deklaration*
Funktionsdeklaration
Bindungs-Spezifizierung

Deklarations-Spezifizierer:
Speicherklassen-Spezifizierer
Typ-Spezifizierer
Funktions-Spezifizierer (C++)
friend (C++)
typedef

Mehrere-Deklarations-Spezifizierer.
<Mehrere-Deklarations-Spezifizierer>Deklarationsspezifizierer

Speicherklassen-Spezifizierer:
auto

register
static
extern

Funktions-Spezifizierer (C++):
inline
virtual

Typ-Spezifizierer:
Einfacher-Typname
Klassen-Spezifizierer
Aufzählungsspezifizierer
Erweiterter-Typ-Spezifizierer
const
volatile

Einfacher-Typname:
Klassenname
typedef-*Name*
char
short
int
long
signed
unsigned
float
double
void

Erweiterter-Typ-Spezifizierer:
Klassenschlüssel Bezeichner
Klassenschlüssel Klassenname
enum *enum-name*

Klassenschlüssel (C++):
class
struct
union

Aufzählungsspezifizierer:
enum <*Bezeichner*>{ <*Aufzählungsliste*>}

Aufzählungsliste:
Aufzähler
Aufzählungsliste, Aufzähler

Aufzähler:
Bezeichner
Bezeichner = *Konstantenausdruck*

Konstantenausdruck:
Bedingter-Ausdruck

Bindungs-Spezifizierung (C++):
extern *String* { <*Deklarationsliste*>}
extern *String Deklaration*

Deklarationsliste:
Deklaration
Deklarationsliste ; *Deklaration*

Beachten Sie bei der folgenden Tabelle, daß es Einschränkungen bei der Anordnung und der Anzahl von Modifizierern und Qualifizierern gibt. Die aufgeführten Modifizierer in der Deklarationssyntax sind in ANSI C und C++ nicht definiert. Diese Modifizier werden detailliert im Abschnitt »Modifizierer« weiter hinten in diesem Kapitel erläutert.

Tabelle 2.2 Deklarationssyntax von Turbo C++

Deklaratorliste:
Initialisierungsdeklarator
Deklaratorliste , *Initialisierungsdeklarator*

Initialisierungsdeklarator:
Deklarator <*Initialisierer*>

Deklarator:
dname
Modifizierer-Liste
Zeiger-Operator Deklarator
Deklarator (*Parameter-Deklarationsliste*) <*cv-Qualifizierer-Liste*>
<*cv-Qualifizierer-Liste*> (*gibt es nur in C++*)
Deklarator [<*Konstantenausdruck*>]
(*Deklarator*)

Modifizierer-Liste:
Modifizierer
Modifizierer-Liste Modifizierer

Modifizierer:
cdecl
pascal
interrupt
near
far
huge

Zeiger-Operator:
* <*cv-Qualifizierer-Liste*>
& <*cv-Qualifizierer-Liste*> (C++)
Klassenname :: * <*cv-Qualifizierer-Liste*> (C++)

cv-Qualifizierer-Liste:
cv-Qualifizierer <*cv-Qualifizierer-Liste*>

cv-Qualifizierer
const
volatile

dname:
Name
Klassenname (C++)
~ *Klassenname* (C++)
typedef-*Name*

Typname:
Typ-Spezifizierer <*Abstrakt-Deklarator*>

Abstrakt-Deklarator:
Zeiger-Operator <*Abstrakt-Deklarator*>
<*Abstrakt-Deklarator*> (*Argument-Deklarationsliste*) <*cv-Qualifizierer-Liste*>
<*Abstrakt-Deklarator*> [<*Konstantenausdruck*>]
(*Abstrakt-Deklarator*)

Argument-Deklarationsliste:
<*Argument-Deklarationen*>
Argument-Deklarationen , ...
<*Argument-Deklarationen*> ... (C++)

Argument-Deklarationen:
Argumentdeklaration
Argument-Deklarationen , *Argumentdeklaration*

Argumentdeklaration:
Mehrere Deklarations-Spezifizierer Deklarator
Mehrere Deklarations-Spezifizierer Deklarator = *Ausdruck* (C++)
Mehrere Deklarations-Spezifizierer <*Abstrakt-Deklarator*>
Mehrere Deklarations-Spezifizierer <*Abstrakt-Deklarator*> = *Ausdruck* (C++)

Funktionsdefinition:
<*Mehrere Deklarations-Spezifizierer*> *Deklarator* <*ctor-Initialisierer*>
Funktionsrumpf

Funktionsrumpf:
Blockanweisung

Initialisierer:
= *Ausdruck*
= { *Initialisierer-Liste* }
(*Ausdrucksliste*) (C++)

Initialisierer-Liste:
Ausdruck
Intialisierer-Liste, Ausdruck
{ *Initialisierer-Liste* <,> }

Externe Deklarationen und Definitionen

Die Speicherklassen-Spezifizierer **auto** und **register** dürfen nicht in externen Deklarationen verwendet werden (vergleichen Sie hierzu den Abschnitt »Übersetzungseinheiten« weiter vorne in diesem Kapitel). Für jeden Bezeichner in einer Übersetzungseinheit, der mit interner Bindung deklariert ist, darf nur eine externe Definition angegeben werden.

Eine externe Definition ist eine externe Deklaration, die ein Objekt oder eine Funktion definiert, also auch Speicherplatz reserviert. Wird ein Bezeichner mit externer Bindung in einem Ausdruck verwendet (außer als Teil des Operanden von **sizeof**), muß es an beliebiger Stelle im Programm genau eine externe Definition für diesen Bezeichner geben.

Turbo C++ erlaubt, daß externe Namen (wie die von Arrays, Strukturen und Varianten) später umdeklariert werden. Auf diese Weise kann früheren Deklarationen Information hinzugefügt werden. Hierzu ein Beispiel:

```
int a[];                // keine Größe
struct mystruct;        // nur ein "Tag", keine Element-
                        // deklaratoren
...
int a[3] = {1, 2, 3};   // Größe und Initialisierung
struct mystruct  {
     int i, j;          // Hinzufügen von Elementdeklaratoren
};
```

In der folgenden Tabelle finden Sie die Syntax für Klassendeklarationen.

Im Abschnitt »Referenzieren« in Kapitel 3 »C++« finden Sie die detaillierte Erläuterung der C++ Referenztypen, die mit Zeigertypen eng verwandt sind.

Tabelle 2.3 Klassendeklarationen in Turbo C++ (nur C++)

Klassen-Spezifizierer:
Klassenkopf { <*Elementliste*> }

Klassenkopf:
Klassenschlüssel <*Bezeichner*><*Basis-Spez*>
Klassenschlüssel Klassenname <*Basis-Spez*>

Elementliste:
Elementdeklaration <*Elementliste*>
Zugriff-Spezifizierer : <*Elementliste*>

Elementdeklaration:
<*Deklarations-Spez*><*Element-Deklaratorliste*>;
Funktionsdefinition <;>
Qualifizierter-Name ;

Element-Deklaratorliste:
Elementdeklarator
Element-Deklaratorliste , Elementdeklarator

Elementdeklarator:
Deklarator <*Abstrakt-Spezifizierer*>
<*Bezeichner*>*: Konstantenausdruck*

Abstrakt-Spezifizierer:
=0

Basis-Spez:
: Basisliste

Basisliste:
Basis-Spezifizierer
Basisliste, Basis-Spezifizierer

Basis-Spezifizierer:
Klassenname
virtual <*Zugriff-Spezifizierer*>*Klassenname*
Zugriff-Spezifizierer <**virtual**>*Klassenname*

Zugriff-Spezifizierer:
private
protected
public

Konvertierungs-Funktionsname:
operator *Konvertierungs-Typname*

Konvertierungs-Typname:
Typ-Spezifizierer <*ptr-Operator*>

ctor-Initialisierer:
: Element-Initialisierer-Liste

Element-Initialisierer-Liste:
Element-Initialisierer
Element-Initialisierer, Element-Initialisierer-Liste

Element-Initialisierer:
Klassenname (<*Argumentliste*>)
Bezeichner (<*Argumentliste*>)

Operator-Funktionsname:
operator *Operator*

Operator: eines der folgenden Zeichen

new	**delete**	**sizeof**						
+	-	*	/	%	^	&	|	~
!	=	<>	+=	-=	*=	/=	%=	^=
&=	|=	<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	--	,	->*	->	()
[]	.*							

Typ-Spezifizierer

Ein *Typ-Spezifizierer* mit einem oder mehreren optionalen Modifizierern legt den Typ des deklarierten Bezeichners fest:

```
int i;                  // deklariert i als vorzeichen-
                        // behaftete Integerzahl
unsigned char ch1, ch2  // deklariert zwei vorzeichenlose
                        // Zeichen
```

Traditionell wird bei fehlendem Typ-Spezifizierer der Typ **signed int** (=**int**) als Voreinstellung angenommen. In C++ gibt es jedoch Situationen, in denen das Fehlen des Typ-Spezifizierers zu syntaktischen Mehrdeutigkeiten führen kann.

Aus diesem Grund sollten in C++ immer alle int-Typ-Spezifizierer angegeben werden.

Typ-Taxonomie

Es gibt vier grundlegende Typ-Kategorien: *void*, *scalar*, *function* und *aggregate*. Die Kategorie *scalar* wird weiterhin unterteilt in arithmetischen, Aufzählungs-, Zeiger- und – nur in C++ – Referenztyp, die Kategorie *aggregate* in Array-, Struktur- Varianten- und – nur in C++ – Klassentyp.

Typen lassen sich auch in *grundlegende* und *abgeleitete* Typen einteilen. Grundlegende Typen sind: **void**, **char**, **int**, **float** und **double** sowie deren **short**-, **long**-, **signed**- und **unsigned**-Varianten. Die abgeleiteten Typen umfassen Zeiger und Referenzen auf andere Typen, Array-, Funktions- und Klassentypen, Strukturen und Varianten.

In C++ kann eine Klasse zum Beispiel eine Anzahl von Objekten mit unterschiedlichen Typen enthalten. Dazu gehören auch Funktionen zur Bearbeitung dieser Objekte und ein Mechanismus zur Steuerung des Zugriffs und der Vererbung aus anderen Klassen.

Aus einem nicht-**void**-Typ *type* können (unter Vorbehalt) abgeleitete Typen wie in Tabelle 2.4 deklariert werden.

Hinweis: Beachten Sie, daß *type& var, type &var* und *type & var* identisch sind.

So würden Sie abgeleitete Typen in einer Klasse deklarieren:

```
class ct  {             // Die Klasse ct speichert einen Zeiger
                        // auf type und eine Funktion, die einen
                        // type-Parameter übernimmt (C++)
    type *ptr;
    public:
    void func(  type*);
}
```

Tabelle 2.4 Typen deklarieren

`type t;`	Ein Objekt vom Typ *type*
`type array[10];`	Zehn *types*: *array*[0] bis *array*[9]
`type *prt;`	*ptr* ist ein Zeiger auf *type*
`type &ref=t;`	*ref* ist eine Referenz auf *type* (C++)
`type func(void);`	*func* liefert einen Wert vom Typ *type*
`void func1 (type t);`	*func1* übernimmt einen Typparameter *type*
`struct st {type t1; type t2};`	Struktur hat zwei *types*

Der Typ void

void ist ein spezieller Typ-Spezifizierer, der anzeigt, daß kein Wert vorhanden ist. Er wird für folgende Zwecke eingesetzt:

- Eine leere Parameterliste in einer Funktionsdeklaration:

```
int func(void);       // func übernimmt keine Argumente
```

Hinweis: C++ behandelt `func()` auf eine spezielle Weise. Näheres dazu erfahren Sie im Abschnitt »Deklarationen und Prototypen« in diesem Kapitel.

- Die deklarierte Funktion liefert keinen Wert:

```
void func(int n);     // kein Rückgabewert
```

- Verwendung als generischer Zeiger: Ein Zeiger auf **void** ist ein generischer Zeiger auf ein beliebiges Objekt:

```
void *ptr;            // ptr kann später als Zeiger  auf ein
                      // beliebiges Objekt gesetzt werden
```

- Verwendung in *typecasting*-Ausdrücken:

```
extern int errfunc(); // liefert einen Fehlercode zurück
...
(void)  errfunc();    // verwirft den Rückgabewert
```

Die grundlegenden Typen

Die *grundlegenden* Typ-Spezifizierer werden aus den folgenden Schlüsselwörtern zusammengesetzt:

char	**double**	**float**	**int**
long	**short**	**signed**	**unsigned**

Hinweis: **signed** und **unsigned** sind Modifizierer, die auf Integertypen angewandt werden können.

Mit diesen Schlüsselwörtern können Sie Integer- und Gleitkommatypen zusammensetzen, die unter dem Oberbegriff *arithmetische Typen* zusammengefaßt werden. In der Include-Datei limits.h sind die Wertebereiche für alle grundlegenden Typen definiert.

Integertypen

char, **short**, **int** und **long** sowie ihre vorzeichenlosen Varianten werden als *Integer*-Datentypen bezeichnet. In der folgenden Tabelle finden Sie die Integertyp-Spezifizierer. Synonyme sind in derselben Zeile aufgelistet.

Tabelle 2.5 Integertypen

char, **signed char**	Synonyme, wenn die Voreinstellung für **char** auf **signed** gesetzt ist
unsigned char	Synonyme, wenn die Voreinstellung für **char** auf **unsigned** gesetzt ist
char, **unsigned char**	
signed char	
int, **signed int**	
unsigned, **unsigned int**	
short, **short int**, **signed short int**	
unsigned short, **unsigned short int**	
long, **long int**, **signed long int**	
unsigned long, **unsigned long int**	

Höchstens eines der Schlüsselwörter **signed** oder **unsigned** kann zusammen mit **char**, **short**, **int** oder **long** verwendet werden. Wenn Sie die Schlüsselwörter **signed** oder **unsigned** alleine einsetzen, bedeuten sie **signed int** beziehungsweise **unsigned int**.

Ist kein **unsigned** vorhanden, wird **signed** verwendet. Eine Ausnahme besteht für **char**. In Turbo C++ kann für **char** sowohl **signed** als auch **unsigned** voreingestellt sein. (Falls Sie es nicht ändern, ist **signed** voreingestellt.) Ist die Voreinstellung **unsigned**, wird `ch` in der Deklaration **char** `ch` als **unsigned** deklariert. Sie müßten mit **signed char** `ch` die Voreinstellung überschreiben. Dasselbe gilt, wenn eine **signed**-Voreinstellung für **char** besteht. Sie müßten dann explizit **unsigned char** `ch` eingeben, um **unsigned char** zu deklarieren.

long und **short** können mit **int** angegeben werden. **long** und **short** alleine bedeuten **long int** und **short int**.

ANSI C definiert nicht die Größe oder interne Repräsentation dieser Typen. Es ist jedoch notwendig, daß **short**, **int** und **long** eine nicht-absteigende Reihenfolge mit »**short** ≤ **int** ≤ **long**« bilden. Alle drei Typen können identisch sein. Das ist dann wichtig, wenn Sie einen portierbaren Programmcode schreiben möchten, der in anderen Umgebungen laufen soll.

In Turbo C++ sind **int** und **short** äquivalent; beide sind 16-Bit-Objekte, während **long** ein 32-Bit-Objekt ist. Die vorzeichenbehafteten Varianten werden im Zweier-Komplement-Format mit dem MSB (Most Significant Bit; also das höchstwertige Bit) als Vorzeichen gespeichert: 0 für positiv, 1 für negativ (dies erklärt die Wertebereiche in Tabelle 1.9). Bei den vorzeichenlosen Varianten werden alle Bits für einen Bereich von 0 - (2^n - 1) verwendet, wobei *n* 8, 16 oder 32 ist.

Gleitkommatypen

Darstellung und Menge der Werte für die Gleitkommatypen sind implementierungsabhängig; das heißt, sie können von jeder C-Implementation frei defi-

niert werden. Turbo C++ verwendet die IEEE-Gleitkomma-Formate (vgl. Kapitel 9 »Spezielle ANSI-Implementierungsstandards«)

float und **double** sind 32- und 64-Bit-Gleitkommatypen. **long** zusammen mit **double** ergibt einen 80-Bit-Gleitkommatyp von hoher Genauigkeit.

In Tabelle 1.9 finden Sie die Speicherbelegung der Gleitkommatypen.

Standardkonvertierungen

Wenn Sie einen arithmetischen Ausdruck, wie $a + b$, haben, wobei a und b zu verschiedenen arithmetischen Typen gehören, führt Turbo C++ vor der Auswertung des Ausdrucks bestimmte interne Konvertierungen durch.

Diese Standardkonvertierungen umfassen auch die Überführung von »niederwertigen« Typen in »höherwertige«, um Genauigkeit und Konsistenz zu gewährleisten. Im folgenden finden Sie die Schritte, die Turbo C++ zur Konvertierung von Operanden in einem arithmetische Ausdruck durchführt:

1. Alle kleinen Integertypen werden so konvertiert, wie Sie es in Tabelle 2.6 sehen. Danach sind die zwei mit einem Operator verbundenen Werte entweder **int** (einschließlich der **long**- und **unsigned**-Modifizierer), **double**, **float** oder **long double**.

2. Wenn einer der beiden Operanden vom Typ **long double** ist, wird der andere Operand in **long double** konvertiert.

3. Wenn einer der beiden Operanden vom Typ **double** ist, wird der andere Operand in **double** konvertiert.

4. Wenn einer der Operanden vom Typ **float** ist, wird der andere Operand in **float** konvertiert.

5. Wenn einer der Operanden vom Typ **unsigned long** ist, wird der andere Operand in **unsigned long** konvertiert.

6. Wenn einer der Operanden vom Typ **long** ist, wird der andere Operand in **long** konvertiert.

7. Wenn einer der Operanden vom Typ **unsigned** ist, wird der andere Operand in **unsigned** konvertiert.

8. In allen anderen Fällen sind beide Operanden vom Typ **int**.

Das Ergebnis des Ausdrucks ist von demselben Typ wie die beiden Operanden.

Tabelle 2.6 Verfahren bei den Standardkonvertierungen

Typ	Konvertierung in	Verfahren
char	**int**	Null oder vorzeichenerweitert (abhängig vom voreingestellten char-Typ)
unsigned char	**int**	Höherwertiges Byte immer Null
signed char	**int**	Vorzeichenerweitert (immer)
short	**int**	Unverändert
unsigned short	**unsigned int**	Unverändert
enum	**int**	Unverändert

Besondere Konvertierungen bei char, int und enum

Die hier erläuterten Konvertierungen sind spezifisch für Turbo C++.

Wird ein vorzeichenbehaftetes Objekt (zum Beispiel eine Variable) einem Integerobjekt zugeordnet, erfolgt eine automatische Erweiterung des Vorzeichens. Objekte vom Typ **signed char** haben immer ein Vorzeichen; Objekte vom Typ **unsigned char** setzen immer das höherwertige Byte auf Null, wenn sie in **int** konvertiert werden.

Bei Konvertierung eines längeren Integertyps in einen kürzeren werden die höherwertigen Bits abgeschnitten und die niederwertigen bleiben unverändert. Bei der Konvertierung eines kürzeren Integertyps in einen längeren wird das Vorzeichen entweder erweitert oder die extra Bits des neuen Wertes werden auf Null gesetzt. Das hängt davon ab, ob der kürzere Typ **signed** oder **unsigned** ist.

Initialisierung

Initialisierer setzen den Anfangswert, der in einem Objekt (Variable, Struktur, Array, usw.) gespeichert wird. Wenn Sie ein Objekt von statischer Dauer nicht initialisieren, wird es per Voreinstellung folgendermaßen initialisiert:

- auf Null gesetzt, wenn es zu einem arithmetischen Typ gehört;
- auf Null gesetzt, wenn es ein Zeigertyp ist.

Wenn das Objekt von automatischer (lokaler) Dauer ist, ist sein Wert unbestimmt. Für Initialisierer gilt folgende Syntax:

Initialisierer
  =*Ausdruck*
  =**{** *Initialisierer-Liste* <**,**> **}**
  (*Ausdrucksliste*)

Initialisierer-Liste
  Ausdruck
  Initialisierer-Liste **,** *Ausdruck*
  { *Initialisierer-Liste* <**,**> **}**

Folgende Regeln gelten für Initialisierer:

1. Die Anzahl der Initialisierer in der Initialisierer-Liste darf nicht größer sein als die Anzahl der Objekte, die initalisiert werden sollen.

2. Nur Arrays unbekannter Größe und Objekte können initialisiert werden.

3. In C (nicht aber in C++) müssen alle Ausdrücke Konstanten sein, wenn sie an einer der folgenden Positionen stehen:

- in einer Initialisierung für ein Objekt mit statischer Dauer;
- in einer Initialisierer-Liste für ein Array, eine Struktur oder eine Variante (Ausdrücke mit **sizeof** sind ebenfalls erlaubt).

4. Wenn eine Deklaration für einen Bezeichner den Gültigkeitsbereich Block hat und der Bezeichner über externe oder interne Bindung verfügt, darf die Deklaration keinen Initialisierer für den Bezeichner haben.

5. Stehen in einer mit geschweiften Klammern eingeschlossenen Liste weniger Initialisierer, als Elemente einer Struktur vorhanden sind, werden die übrigbleibenden Elemente implizit wie Objekte mit statischer Speicherplatzdauer initialisiert.

Skalare Typen werden mit einem einzigen Ausdruck initialisiert, der optional in geschweiften Klammern stehen kann. Der Anfangswert des Objekts entspricht dem des Ausdrucks; in Bezug auf Typ und Umwandlungen treffen dieselben Beschränkungen zu wie bei einfachen Zuweisungen.

Bei Varianten initialisiert ein in geschweiften Klammern stehender Initialisierer das Element, das als erstes in der Liste der Variantendeklaration steht. Strukturen und Varianten von automatischen Speicherklassen erfordern einen der folgenden Initialisierer:

- eine Initialisierer-Liste, wie sie im folgenden Abschnitt beschrieben wird;
- einen Ausdruck mit kompatiblem Varianten- oder Strukturtyp. In diesem Fall entspricht der Anfangswert des Objektes dem des Ausdrucks.

Array, Strukturen und Varianten

Sie können Arrays und Strukturen (zum Beispiel bei der Deklaration) mit einer in geschweiften Klammern stehenden Liste von Initialisierern für Elemente des in Frage kommenden Objekts initialisieren. Die Initialisierer werden in aufsteigender Reihenfolge der Array-Indizes oder Elemente angegeben. Varianten initialisieren Sie mit einem in geschweiften Klammern stehenden Initialisierer für das erste Element der Variante. So könnten Sie etwa ein Array *days* deklarieren, das zählen soll, wie oft jeder Wochentag in einem Monat vorkommt (und dabei unterstellen, daß jeder Tag wenigstens einmal vorkommt):

```
int days[7] = { 1, 1, 1, 1, 1, 1, 1 }
```

Die folgenden Regeln gelten bei der Initialisierung von Zeichen- und »wide character«-Arrays:

1. Zeichen-Arrays können mit einem String-Literal (optional in geschweiften Klammern) initialisiert werden. Jedes Zeichen des Strings (einschließlich der abschließenden Null) initialisiert nacheinander ein Element in dem Array. So können Sie zum Beispiel folgende Deklaration abgeben:

```
char name[] = { "Unknown" };
```

In dieser Deklaration wird ein Array mit acht Elementen definiert. Die Elemente sind 'U' (für *name*[0]), 'n'(für *name*[1]), etc. (einschließlich einer abschließenden Null).

2. Sie können ein »wide character«-Array (das kompatibel mit *wchar_t* ist) mit einem »wide String«-Literal initialisieren, das in geschweiften Klammern stehen kann. Wie bei Zeichen-Arrays initialisieren die Codes des »wide String«-Literals nacheinander die Elemente des Array.

Im folgenden sehen Sie ein Beispiel für die Initialisierung einer Struktur:

```
struct mystruct  {
    int i;
```

```
    char str[21];
    double d;
} s = { 20, "Borland", 3.141 };
```

Komplexe Elemente einer Struktur, wie Arrays oder Strukturen, können mit entsprechenden Ausdrücken innerhalb geschachtelter geschweifter Klammern initialisiert werden. Sie können die Klammern weglassen, dafür müssen Sie sich dann an bestimmte Regeln halten (wir raten Ihnen davon ab).

Einfache Deklarationen

Einfache Deklarationen von Variablenbezeichnern haben folgendes Muster:

Datentyp var1 <=init1>, var2 <=init2>, ...;

wobei *var1*, *var2*, ... eine beliebige Folge verschiedener Bezeichner ist, die zusätzlich Initialisierer haben kann. Alle Variablen sind vom Typ *Datentyp*; beispielsweise erzeugt

```
int x = 1, y = 2;
```

zwei Integervariable, *x* und *y* (und initialisiert sie mit den Werten 1 und 2).

Es handelt sich hierbei um eine definierende Deklaration, das heißt, es erfolgt eine Speicherzuweisung und die optionalen Initialisierer werden angewendet.

Als Initialisierer für ein automatisches Objekt kann jeder zulässige Ausdruck verwendet werden, der zu einem zuweisungskompatiblen Wert für den Typ der betreffenden Variablen ausgewertet wird. Die Initialisierer für statische Objekte müssen Konstanten oder Konstantenausdrücke sein.

In C++ kann als Initialisierer für ein statisches Objekt jeder Ausdruck mit Konstanten und bereits deklarierten Variablen und Funktionen verwendet werden.

Speicherklassen-Spezifizierer

Eine Deklaration muß einen *Speicherklassen-Spezifizierer* (oder Typ-Spezifizierer) enthalten. Folgende Speicherklassen-Spezifizierer stehen zur Verfügung:

auto	**register**	**typedef**
extern	**static**	

auto

Der Speicherklassen-Spezifizierer **auto** wird nur bei Variablendeklarationen mit lokalem Gültigkeitsbereich verwendet. **auto** impliziert lokale (automatische) Dauer, da diese aber schon standardmäßig für alle Variablendeklarationen mit lokalem Gültigkeitsbereich gilt, wird dieser Spezifizierer selten verwendet.

extern

Der Speicherklassen-Spezifizierer **extern** kann in Deklarationen für Funktionen und Variablen mit dem Gültigkeitsbereich Datei und Block verwendet werden, um eine externe Bindung zu definieren. Bei Variablen mit dem Gültigkeitsbereich Datei wird standardmäßig die Speicherklasse **extern** verwendet. Wird **extern** für Variablen eingesetzt, heißt das, daß die Variable von statischer Dauer ist. (Eine Funktion ist immer von statischer Dauer.)

Im Abschnitt »Namenserweiterung« weiter vorne in diesem Kapitel finden Sie Informationen über die Verwendung von **extern** zur Verhinderung von Namensergänzung, wenn C- und C++Code kombiniert wird.

register

Der Speicherklassen-Spezifizierer **register** kann nur bei Deklarationen von lokalen Variablen und Funktionsparametern verwendet werden. **register** impliziert **auto**, weist aber den Compiler an, die Variable wenn möglich in einem Register zu speichern. Dadurch läßt sich oft die Größe des Programms verringern und dessen Ausführungsgeschwindigkeit steigern.

Da Turbo C++die Positionierung von Variablen in Registern selbst durchführen kann, ist die Verwendung des Schlüsselworts **register** nur sehr selten erforderlich. Durch Markieren von *Automatisch* im Menü *Optionen/Compiler/Optimierungen* weisen Sie Turbo C++an, Register zuzuweisen – auch dann, wenn der Speicherklassen-Spezifizierer **register** nicht eingesetzt wird.

static

Der Speicherklassen-Spezifizierer **static** kann in Deklarationen für Funktionen und Variablen mit dem Gültigkeitsbereich Datei und Block verwendet werden, um eine interne Bindung anzuzeigen. Variablen der Speicherklasse **static** haben statische Dauer. Wenn keine Konstruktoren oder explizite Initialisierer verwendet werden, werden statische Variablen beim Start des Programms auf den Wert Null gesetzt.

In C++ hat ein statisches Datenelement einer Klasse für alle Instanzen der Klasse denselben Wert. Eine statische Elementfunktion kann unabhängig davon aufgerufen werden, ob eine Instanz der Klasse existiert.

typedef

Das Schlüsselwort **typedef** zeigt an, daß kein Objekt deklariert, sondern ein neuer Datentyp-Spezifizierer definiert wird. Wir wollen aber hier auf **typedef** eingehen, da syntaktische Ähnlichkeiten mit den Speicherklassen-Spezifizierern vorhanden sind.

```
static  long int biggy;
typedef long int BIGGY;
```

Die erste Deklaration erzeugt ein Objekt vom Typ **long int**, also ein 32-Bit-Objekt mit statischer Dauer namens *biggy*. Mit der zweiten Deklaration wird der Bezeichner BIGGY als neuer Typ-Spezifizierer eingeführt, jedoch kein Laufzeit-Objekt erzeugt. BIGGY kann in allen folgenden Deklarationen verwendet werden, in denen ein Typ-Spezifizierer zulässig ist. Dazu ein Beispiel:

```
extern BIGGY Gehalt;
```

hat denselben Effekt wie

```
extern long int Gehalt;
```

In diesem Beispiel könnte dasselbe Resultat auch mit `#define BIGGY long int` erzielt werden. Mit komplexeren **typedef**-Anwendungen werden aber Ergebnisse erzielt, die mit textuellen Ersetzungen nicht mehr möglich wären.

Wichtig: **typedef** erzeugt keinen neuen Datentyp, sondern führt lediglich Synonyme (einfachere Namen) für einen existierenden Typ ein. **typedef** eignet sich daher besonders zur Vereinfachung komplexer Deklarationen:

```
typedef double (*PFD) ();
PFD array_pfd[10];
/* array_pfd ist ein Array mit 10 Zeigern auf
   Funktionen, die einen double-Wert zurückliefern */
```

typedef-Bezeichner können nicht mit anderen Datentyp-Spezifizierern verwendet werden:

```
unsigned BIGGY Gehaltsklasse;          /* ILLEGAL */
```

Modifizierer

Neben den Schlüsselwörtern zur Spezifizierung einer Speicherklasse können in einer Deklaration *Modifizierer* eingesetzt werden, um die zum Bezeichner/Objekt gehörige Speicherbelegung zu ändern. In Tabelle 2.7 sind die in Turbo C++ verfügbaren Modifizierer aufgelistet

Der Modifizierer const

Der Modifizierer **const** verhindert Zuweisungen an ein Objekt und andere Nebeneffekte, wie Inkrementierung und Dekrementierung. Ein **const**-Zeiger kann nicht modifiziert werden, die Veränderung des Objekts, auf das er zeigt, ist dagegen möglich. Wird der Modifizierer **const** alleine verwendet, entspricht er dem Typ **const int**.

Dazu einige Beispiele:

```
const   float   pi      = 3.1415926;
const           maxint  = 32767;
char    *const  str     = "Hello, world";
                                /* Ein konstanter Zeiger */
char    const   *str2   = "Hello, world";
                                /* Ein Zeiger auf ein
                                   konstantes char */
```

Nach diesen Deklarationen sind folgende Befehle unzulässig:

```
pi  = 3.0;                      /* Zuordnung eines Wertes an
                                   eine Konstante */
i   = maxint++;                 /* Inkrementierung einer Konstante */
str = "Hi, there!";             /* Zuweisung einer anderen
                                   Adresse */
```

Der Funktionsaufruf `strcpy (str,"Hi, there!")` wäre allerdings legal, da er den Text *»Hi, there!«* zeichenweise in den Speicherbereich kopiert, auf den *str* zeigt.

In C++verbirgt **const** das **const**-Objekt und verhindert eine externe Bindung; diese kann mit dem Modifizierer **extern const** erreicht werden. Ein Zeiger auf einen **const**-Wert darf nicht einem Zeiger auf ein nicht-**const** zugewiesen werden, da sonst eine Zuweisung an den **const**-Wert mit Hilfe des nicht-**const** Zeigers möglich wäre. Hierzu ein Beispiel:

```
char *str3 = str2 /* nicht erlaubt */
```

Es können nur **const**-Elementfunktionen für ein **const**-Objekt aufgerufen werden.

Tabelle 2.7 Turbo C++Modifizierer

Modifizierer	Verwendung mit	Funktion
const	nur Variablen	Verhindert die Änderung des Objekts.
volatile	nur Variablen	Verhindert die Registerzuweisung und andere Optimierungsmaßnahmen. Warnt den Compiler, daß das Objekt während der Auswertung von außen geändert werden kann.
Erweiterungen in Turbo C++		
cdecl	Funktionen	Erzwingt die C-Konvention für die Übergabe von Argumenten. Betrifft den Linker und beim Linken vergebene Namen.
cdecl	Variablen	Erzwingt den führenden Unterstrich und die Unterscheidung von Groß- und Kleinbuchstaben bei globalen Bezeichnern.
pascal	Funktionen	Erzwingt die Pascal-Konvention für die Übergabe von Argumenten. Betrifft den Linker und beim Linken vergebene Namen.
pascal	Variablen	Erzwingt die Gleichbehandlung von Groß- und Kleinbuchstaben bei globalen Bezeichnern ohne führenden Unterstrich.
interrupt	Funktionen	Die Funktion wird mit zusätzlichen Befehlen zur Sicherung der Register versehen, die beim Programmieren von Interrupt-Handlern erforderlich sind.
near, far, huge	Zeigervariablen	Überschreibt den voreingestellten Zeigertyp, der vom aktuellen Speichermodell abhängt.
_cs, _ds, _es, _seg, _ss	Zeigervariablen	Segmentzeiger
near, far, huge	Funktionen	Überschreibt den voreingestellten Funktionstyp, der vom aktuellen Speichermodell abhängig ist.
near, far	Variablen	Steuert die Plazierung des Objekts im Speicher.
_export	Funktionen/ Klassen	Teilt dem Compiler mit, welche Funktionen oder Klassen exportiert werden sollen.

Modifizierer	Verwendung mit	Funktion
_loadds	Funktionen	Sorgt dafür, daß DS auf das aktuelle Datensegment zeigt.
_saveregs	Funktionen	Speichert alle Registerwerte (außer Rückgabewerten) während der Ausführung der Funktion.

Der Modifzierer interrupt

Der Modifizierer **interrupt** ist Turbo C++ spezifisch und ausschließlich für Funktionen vorgesehen, die über die Interrupt-Vektoren des 8086/8088 aufgerufen werden. Turbo C++compiliert eine als **interrupt** deklarierte Funktion mit zusätzlichen Befehlen an ihrem Anfang und Ende, die zum Sichern der Register AX, BX, CX, DX, SI, DI, ES und DS dienen. Die anderen Register (BP, SP, SS, CS und IP) werden durch den Funktionsaufruf bzw. durch den Interrupt-Handler selbst auf dem Stack zwischengespeichert. Die Funktion verwendet zur Rückkehr eine **iret**-Anweisung, damit sie zur Behandlung von Hardware- und Software-Interrupts verwendet werden kann. Im folgenden sehen Sie ein Beispiel für eine typische **interrupt**-Definition:

```
void interrupt myhandler()
{
   ...
}
```

Interrupt-Funktionen sollten immer als **void** deklariert werden und können in allen Speichermodellen deklariert werden. In allen Modellen – außer *Huge* – wird DS auf das Programm-Datensegment gesetzt, beim Modell *Huge* auf das Modul-Datensegment.

Der Modifizierer volatile

Der Modifizierer **volatile** bestimmt, daß das Objekt geändert werden kann, und zwar nicht nur durch das Programm, sondern auch von »außerhalb« (das heißt durch eine Interrupt-Routine oder einen I/O-Port). Die Deklaration als **volatile** warnt den Compiler davor, Annahmen über den Wert des betreffenden Objekts zu machen, da er sich (theoretisch) jederzeit ändern kann. Mit **volatile** deklarierte Variablen werden nie als Registervariablen behandelt.

```
volatile int ticks;
interrupt timer()
{
   ticks++,
}
wait(int interval)
{
   ticks = 0;
   while (ticks < interval);   //Warten
}
```

Diese Routinen implementieren (vorausgesetzt, daß *timer* in geeigneter Weise dem Hardware-Timer-Interrupt zugeordnet wurde) eine Pausenzeit, die vom

Argument *interval* festgelegt wird. Ein optimierungsfreudiger Compiler würde unter Umständen den Wert von *ticks* für den Test nicht in die **while**-Schleife laden, da die Schleife den Wert von *ticks* nicht verändert.

Hinweis: In C++ hat **volatile** eine spezielle Bedeutung für Klassenelementfunktionen. Wenn ein Objekt als **volatile** deklariert wurde, können nur seine **volatile**-Elementfunktionen verwendet werden.

Die Modifizierer cdecl und pascal

Turbo C++ ermöglicht Ihren Programmen den Aufruf von Routinen, die in anderen Sprachen geschrieben sind (und umgekehrt). Bei der Arbeit mit verschiedenen Sprachen müssen Sie zwei Probleme lösen: die Bezeichner und die Übergabe von Parametern.

In Turbo C++ werden alle globalen Bezeichner in unveränderter Schreibweise (Groß- oder Kleinbuchstaben bzw. eine Kombination davon) mit vorangestelltem Unterstrich (_) gespeichert, außer die die Option *Unterstriche erzeugen* ist im Dialogfenster des Menüs *Optionen/Compiler/Erweiterte Code-Generierung* ausgeschaltet.

Im Abschnitt »Bindung« weiter vorne in diesem Kapitel erfahren Sie, wie Sie unter Verwendung von **extern** in C++ Programmen auf C-Namen Bezug nehmen können.

pascal In Pascal werden die globalen Bezeichner nicht in ihrer ursprünglichen Schreibweise und mit vorangestelltem Unterstrich gespeichert. In Turbo C++ kann jeder Bezeichner als **pascal** deklariert werden. Derartige Bezeichner werden in Großbuchstaben und ohne einen vorangestellten Unterstrich gespeichert. (Handelt es sich bei dem Bezeichner um eine Funktion, wird auch die Reihenfolge der Parameterübergabe beeinflußt; vgl. Sie den Abschnitt »Modifizierer für Funktionstypen« im folgenden).

Hinweis: Wenn Sie im Menü *Optionen/Compiler/Entry-/Exitcode* die Pascal-Aufrufkonvention wählen, werden sämtliche Funktionen (und die Zeiger auf diese Funktionen) als **pascal** behandelt.

Der Modifizierer **pascal** ist eine spezielle Erweiterung in Turbo C++ und für Funktionen (und Zeiger auf Fuktionen) vorgesehen, die das Pascal-Format für die Parameterübergabe verwenden. Als **pascal** deklarierte C-Funktionen können auch von anderen C-Funktionen aufgerufen werden, vorausgesetzt, die aufrufende Funktion »weiß«, daß die aufgerufene vom Typ **pascal** ist.

```
pascal putnums(int i, int j, int k)
{
   printf("Die Zahlen sind: %   d, %d und %d\n", i,j,k);
}
```

Im Gegensatz zu Funktionen wie *printf* kann eine Funktion vom Typ **pascal** nicht mit einer variablen Anzahl von Argumenten arbeiten – die Angabe von einer Ellipse (...) ist deshalb nicht zulässig.

Wichtig: Die meisten Funktionen der Windows-API sind vom Typ **pascal**.

cdecl Wenn Sie nach einer Compilierung mit der Pascal-Aufrufkonvention (*Optionen/Compiler/Entry-/Exitcode*) sicherstellen wollen, daß für bestimmte Be-

zeichner (speziell C-Bezeichner aus einer anderen Datei) die Schreibweise beibehalten und der führende Unterstrich verwendet wird, können Sie diese Bezeichner als **cdecl** deklarieren. (Dies hat auch Auswirkungen auf die Parameterübergabe.)

Ebenso wie **pascal** ist auch der Modifizierer **cdecl** Turbo C++ spezifisch und wird mit Funktionen und Zeigern auf Funktionen verwendet. Er überschreibt die die Compiler-Direktive *Optionen/Compiler/Entry-/Exitcode* und ermöglicht, daß eine Funktion als reguläre C-Funktion aufgerufen werden kann. Soll das im Abschnitt über **pascal** gezeigte Programm mit der *Pascal*-Option compiliert und *printf* verwendet werden, ist das mit der folgenden Deklaration möglich:

```
extern cdecl printf();
void putnums(int i, int j, int k);

cdecl main()
{
   putnums(1,4,9);
}

putnums(int i, int j, int k)
{
   printf("Die Zahlen sind: %   d, %d und %d\n", i,j,k);
}
```

Beim Compilieren eines Programms mit dem IDE-Menü *Optionen/Compiler/Entry-/Exitcode*) müssen alle Funktionen aus der Laufzeitbibliothek als **cdecl** deklariert sein. Sehen Sie sich die Include-Dateien (zum Beispiel stdio.h) an – Sie werden bemerken, daß sämtliche Funktionen explizit als **cdecl** deklariert sind.

Zeiger-Modifizierer

Turbo C++ verfügt über acht Modifizierer, die den Indirektionsoperator (*) und damit die Datenzeiger beeinflussen: *near*, *far*, *huge*, **_cs**, *_ds*, *_es*, *_seg* und *_ss*.

Die Compilierung kann in C mit verschiedenen Speichermodellen erfolgen. Das verwendete Modell bestimmt unter anderem auch das interne Format der Zeiger. Wird ein »kleines« Modell eingesetzt (*Small* oder *Medium*), enthalten alle Datenzeiger einen 16-Bit-Offset aus dem Datensegment-Register (DS). Verwenden Sie ein »großes« Modell (*Compact*, *Large*), werden Datenzeiger mit 32 Bits verwendet, die sowohl eine Segmentadresse als auch einen Offset enthalten. Mit Hilfe von Zeiger-Modifizierern wird die Größe eines Zeigers unabhängig vom gewählten Speichermodell festgelegt.

Modifizierer für Funktionstypen

Die Modifizierer **near**, **far** und **huge** können auch auf Funktionen, auf Zeiger auf Funktionen und auf Datenzeiger angewendet werden. Zur Modifizierung von Funktionen stehen außerdem noch **_export**, **_loadds** und **_saveregs** zur Verfügung.

Die Funktions-Modifizierer **near**, **far** und **huge** können mit **cdecl** und **pascal**, nicht aber mit **interrupt**, kombiniert werden.

Funktionen vom Typ **huge** sind sinnvoll, um eine Schnittstelle zu einem Assemblerprogramm herzustellen, das nicht dieselbe Speicherbelegung verwendet wie Turbo C++.

Eine Nicht-**interrupt**-Funktion kann als **near**, **far** oder **huge** deklariert werden. Auf diese Weise ist es möglich, die Standardeinstellung des aktuellen Speichermodells zu überschreiben.

Eine **near**-Funktion verwendet **near**-Aufrufe, eine **far**- oder **huge**-Funktion **far**-Aufrufe.

In den Speichermodellen *Small* und *Compact* verwendet eine Funktion ohne Typangabe standardmäßig den Typ **near**. In den Speichermodellen *Medium* und *Large* wird in derartigen Fällen der Typ **far**, im Modell *Huge* der Typ **huge** verwendet.

Eine **huge**-Funktion ist bis auf folgende Ausnahme mit einer **far**-Funktion identisch: Beim Einsprung in eine **huge**-Funktion wird das DS-Register auf die Datensegmentadresse des Quellmoduls gesetzt; bei einer **far**-Funktion bleibt es dagegen unverändert.

Der Modifizierer **_export** bestimmt, daß eine Funktion aus Windows exportierbar ist. Er wird in einer ausführbaren Datei (wenn Sie nicht Smart Callbacks verwenden) oder einer DLL eingesetzt. In Kapitel 8 »Erstellen einer Windows-Anwendung« finden Sie Erläuterungen zu diesem Thema. In DOS-Programmen hat **_export** keine Bedeutung.

Der Modifizierer **_loadds** zeigt an, daß (wie bei einer **huge**-Funktion) das DS-Register gesetzt werden soll, impliziert aber keine **near**- oder **far**-Aufrufe. **_loadds far** ist demnach gleichbedeutend mit **huge**.

Der Modifizierer **_saveregs** sorgt dafür, daß die Funktion alle Registerwerte sichert und vor der Rückgabe wiederherstellt (ausgenommen explizite Rückgabewerte, die in Registern wie AX oder DX zurückgegeben werden).

Die Modifizierer **_loadds** und **_saveregs** sind beim Schreiben von »Low-Level«-Schnittstellenroutinen (zum Beispiel von Maustreiberroutinen) besonders hilfreich.

Komplexe Deklarationen und Deklaratoren

Einfache Deklarationen bestehen aus einer Liste mit Bezeichnern, die durch Kommata begrenzt sind, gefolgt von optionalen Speicherklassen- oder Typ-Spezifizierern oder sonstigen Modifizierern.

Tabelle 2.1 behandelt die Deklarationssyntax. Die Definition betrifft sowohl Bezeichner als auch Funkionsdeklarationen.

Eine komplexe Deklaration verwendet eine Liste von kommabegrenzten Deklaratoren, gefolgt von verschiedenen Spezifizierern und Modifizierern. Innerhalb jedes Deklarators gibt es nur einen Bezeichner – den, der gerade deklariert wird. Jeder Deklarator in der Liste ist dem vorangestellten Speicherklassen- und Typ-Spezifizierer zugeordnet.

Das Format eines Deklarators zeigt an, wie der deklarierte *dname* interpretiert werden soll, wenn er in einem Ausdruck verwendet wird. Im folgenden Beispiel gehen wir davon aus, daß *type* ein beliebiger Typ und *speicherklassen-spezifizierer*

ein beliebiger Speicherklassen-Spezifizierer ist, *D1* und *D2* sind zwei beliebige Deklaratoren. Die folgende Deklaration

speicherklassen-spezifizierer type D1, D2;

zeigt an, daß jedes Vorkommen von *D1* oder *D2* in einem Ausdruck als Objekt des Typs *type* und der Speicherklasse *speicherklassen-spezifizierer* gewertet wird. Der Typ des im Deklarator eingebetteten Bezeichners *dname* besteht aus einer Angabe, die *type* enthält, wie zum Beispiel »*type*«, »Zeiger auf *type*«, »Array von *type*«, »Funktion, die *type* zurückliefert« oder »Zeiger auf Funktion, die *type* zurückliefert«. Beispielsweise kann mit den Deklarationen

```
int n, nao[], naf[3], *pn, *apn[], (*pan)[], &nr=n;
int f(void), *fnp(void), (*pfn) (void);
```

jeder Deklarator in Ausdrücken, in denen man ein einzelnes **int**-Objekt einsetzen könnte, als R-Wert (in bestimmten Fällen auch als L-Wert) verwendet werden.

Die Typen der eingebetteten Bezeichner werden folgendermaßen von ihren Deklaratoren abgeleitet:

Tabelle 2.8 Komplexe Deklarationen

Deklarator-Syntax	implizierter Typ von *name*	Beispiel
type name;	*type*	`int count;`
type name[];	(offenes) Array von *type*	`int count[];`
type name[3];	(festes) Array mit drei *type*-Elementen (*name*[0], *name*[1] und *name*[2])	`int count[3];`
type **name*;	Zeiger auf *type*	`int *count;`
type **name*[];	(offenes) Array von Zeigern auf *type*	`int *count[];`
type *(*name*[]);	wie oben	`int *(count []);`
type (**name*) [];	Zeiger auf ein (offenes) Array von *type*	`int (*count) [];`
type &*name*;	Referenz auf *type* (nur C++)	`int &count;`
type name();	Funktion, die *type* zurückliefert	`int count();`
type **name*();	Funktion, die Zeiger auf *type* zurückliefert	`int *count();`
type *(*name*());	wie oben	`int *(count ());`
type (**name*) ();	Zeiger auf Funktion, die *type* zurückliefert	`int (*count) ();`

Beachten Sie, daß in (**name*)[] und (**name*)() runde Klammern benötigt werden, da der Array-Deklarator [ ] und der Funktionsdeklarator ( ) gegenüber dem Zeigerdeklarator * Vorrang haben. Die runden Klammern in *(*name*[]) sind optional.

Zeiger

Zeiger lassen sich in zwei Kategorien einordnen: Zeiger auf Objekte und Zeiger auf Funktionen. Beide Zeigertypen sind spezielle Objekte, die Speicheradressen enthalten.

Die beiden Zeigerklassen haben unterschiedliche Eigenschaften, Zwecke und Regeln für die Handhabung, obwohl sie teilweise in denselben Turbo C++ Operationen auftreten können. Generell kann gesagt werden, daß Zeiger auf Funktionen für den Zugriff auf Funktionen und für die Übergabe von Funktionen als Argumente an andere Funktionen verwendet werden. Berechnungen mit Zeigern auf Funktionen sind nicht zulässig. Andererseits werden Zeiger auf Objekte inkrementiert oder dekrementiert, wenn Arrays oder komplexe Datenstrukturen im Speicher abgesucht werden.

Obwohl Zeiger Zahlen beinhalten, die fast dieselben Eigenschaften haben wie vorzeichenlose Integerzahlen, gelten für diese eigene Regeln und Restriktionen in Bezug auf Zuweisungen, Konvertierung und Arithmetik. Die Beispiele in den nächsten Abschnitten illustrieren diese Regeln und Restriktionen.

Hinweis: Im Abschnitt »Unäre Operatoren« weiter hinten in diesem Kapitel finden Sie Erläuterungen zu Referenzierung und Dereferenzierung.

Zeiger auf Objekte

Ein Zeiger des Typs »Zeiger auf Objekt vom Typ *type*« enthält die Adresse eines Objekts (d.h er zeigt auf das Objekt) vom Typ *type*. Da Zeiger selbst Objekte sind, können auch Zeiger auf Zeiger zeigen (usw.). Arrays, Strukturen, Varianten und Klassen sind weitere Objekte, die gewöhnlich über Zeiger angesprochen werden.

Die Größe von Zeigern auf Objekte hängt vom Speichermodell und der Größe und Position Ihrer Datensegmente ab, möglicherweise beeinflußt von optionalen Zeiger-Modifizierern (vgl. dazu den Abschnitt »Modifizierer« weiter vorne in diesem Kapitel).

Zeiger auf Funktionen

Einen Zeiger auf eine Funktion kann man sich als Adresse – normalerweise in einem Code-Segment – vorstellen, an deren Position der ausführbare Code für diese Funktion steht. Dies ist die Adresse, der die Kontrolle übergeben wird, wenn diese Funktion aufgerufen wird. Die Größe und Position Ihres Code-Segments wird vom Speichermodell bestimmt, das seinerseits die Größe der Funktionszeiger festlegt, die für den Aufruf der Funktionen notwendig sind.

Ein Zeiger auf eine Funktion hat den Typ »Zeiger auf eine Funktion, die *type* zurückliefert«, wobei *type* der Rückgabetyp der Funktion ist.

In C++, in dem es eine strengere Typenprüfung gibt, hat ein Zeiger auf eine Funktion den Typ »Zeiger auf eine Funktion, deren Argumente den Typ *type* haben, und die *type* liefert«. Tatsächlich hat auch in C eine Funktion, die mit Argumenttypen definiert ist, diesen engeren Typ. Beispiel:

```
void (*func) ();
```

In C ist dies ein Zeiger auf eine Funktion, die nichts liefert; in C++ ein Zeiger auf eine Funktion, die keine Argumente hat und nichts liefert.

Im Beispiel:

```
void (*func) (int);
```

ist **func* ein Zeiger auf eine Funktion, die ein **int** als Argument hat und nichts liefert.

Zeiger-deklarationen

Ein Zeiger muß als auf einen bestimmten Typ zeigend deklariert werden, auch wenn dieser Typ **void** ist (also als Zeiger auf irgendein Objekt). Sobald ein Zeiger einmal deklariert ist, kann man ihn auch auf ein Objekt mit einem anderen Typ zeigen lassen. In Turbo C++ können Sie Zeiger ohne Typumwandlung (type-casting) neu zuordnen. Sie erhalten dabei jedoch eine Warnung vom Compiler, außer der Zeiger war ursprünglich als Zeiger auf **void** deklariert. In C, aber nicht in C++, können Sie einen *void**-Zeiger einem Nicht-*void**-Zeiger zuweisen.

Wenn *type* ein vordefinierter oder benutzerdefinierter Typ ist, einschließlich **void**, dann legt die Deklaration

```
type *ptr;      /* Achtung -- nicht-initialisierter Zeiger */
```

ptr als Zeiger vom Typ »Zeiger auf *type*« fest. Alle Regeln für Gültigkeitsbereich, Dauer und Sichtbarkeit treffen auf das gerade deklarierte *ptr*-Objekt zu.

Hinweis: Sie müssen Zeiger initialisieren, bevor Sie sie verwenden.

Ein Null-Zeigerwert ist eine Adresse, die sich garantiert von jedem gültigen Zeiger in einem Programm unterscheidet. Die Zuweisung der Integerkonstante 0 ordnet einem Zeiger einen Null-Zeigerwert zu.

Aus Gründen der besseren Lesbarkeit kann das Kürzel NULL (definiert in den Include-Dateien der Standardbibliothek, zum Beispiel in stdio.h) verwendet werden. Alle Zeiger können erfolgreich auf Gleichheit oder Ungleichheit zu NULL überprüft werden.

Der Typ »Zeiger auf **void**« darf nicht mit dem Null-Zeiger verwechselt werden. Die Deklaration

```
void *vprt;
```

deklariert *vptr* als generischen Zeiger, dem jeder »Zeiger auf *type*«-Wert, einschließlich Null, zugewiesen werden kann.

Zuweisungen ohne geeignete Typumwandlung zwischen einem »Zeiger auf *type1*« und einem »Zeiger auf *type2*« – wobei *type1* und *type2* unterschiedliche Typen sind – können eine Compiler-Warnung oder einen Fehler hervorrufen. Wenn *type1* eine Funktion und *type2* keine Funktion ist (oder umgekehrt), sind Zeigerzuweisungen nicht erlaubt. Ist *type1* ein Zeiger auf **void**, ist eine Typumwandlung nicht notwendig, ist *type2* ein Zeiger auf **void**, so ist in C ebenfalls keine Typumwandlung erforderlich.

Zuordnungsrestriktionen treffen auch auf Zeiger mit unterschiedlichen Größen zu (**near**, **far** und **huge**). Sie können problemlos einem größeren Zeiger einen kleineren zuweisen, in der umgekehrten Richtung müßten Sie jetzt eine explizite Typumwandlung vornehmen.

Betrachten Sie folgendes Beispiel:

```
char near *ncp;
char far  *fcp;
char huge *hcp;
fcp = ncp;                  // zulässig
hcp = fcp;                  // zulässig
fcp = hcp;                  // nicht zulässig
ncp = fcp;                  // nicht zulässig
ncp = (char near*)fcp;      // jetzt zulässig
```

Zeiger und Konstanten

Ein Zeiger oder das Objekt, auf das gezeigt wird, kann mit dem **const**-Modifizierer deklariert werden. Einem Objekt, das als **const** deklariert ist, kann nichts zugewiesen werden. Es ist auch nicht zulässig, einen Zeiger anzulegen, der diese Regel verletzt.

Betrachten Sie die folgenden Beispiele:

```
int i;                      // i ist ein int

int *pi;                    // pi ist ein Zeiger auf
                            // int (uninitialisiert)

int * const cp = &i;        // cp ist ein konstanter
                            // Zeiger auf int

const int ci = 7;           // ci ist ein konstanter int

const int * pci;            // pci ist ein Zeiger auf
                            // ein konstantes int

const int * const cpc = &ci; // cpc ist ein konstanter Zeiger auf
                            // ein konstantes int
```

Folgende Zuweisungen sind zulässig:

```
i = ci;                     // Weist const-int einem int zu

*cp = ci;                   // Weist const-int einem Objekt zu, auf
                            // das ein konstanter Zeiger zeigt

++pci;                      // Inkrementiert einen Zeiger auf const

pci = cpc;                  // Weist einen konstanten Zeiger auf
                            // const einem Zeiger auf const zu
```

Folgende Zuweisungen sind unzulässig:

```
ci = 0;                     // Zuweisung an ein const-int nicht
                            // erlaubt

ci--;                       // const-int kann nicht geändert werden
```

```
*pci = 3;           // Zuweisung zu einem Objekt, auf das ein
                    // Zeiger-auf-const zeigt, ist nicht erlaubt

cp = &ci;           // Zuweisung zu einem konstanten Zeiger
                    // ist nicht erlaubt, auch wenn der Wert
                    // nicht verändert wird

cpc++;              // Nein -- konstanter Zeiger kann nicht
                    // geändert werden

pi = pci;           // Wenn dies erlaubt wäre, könnten Sie
                    // dem konstanten Wert *pci über *pi
                    // einen Wert zuweisen
```

Ähnliche Regeln gelten für den Modifizierer **volatile**. Beachten Sie, daß **const** und **volatile** als Modifizierer desselben Bezeichners stehen können.

Zeiger-arithmetik

Die Zeigerarithmetik beschränkt sich auf Addition, Subtraktion und Vergleich. Arithmetische Operationen bei Zeigern vom Typ »Zeiger auf *type*« berücksichtigen automatisch die Größe des Typs **type**, d.h die Anzahl der Bytes, die für das Speichern des *type*-Objekts notwendig sind.

Hinweis: Die intern ausgeführte Zeigerarithmetik richtet sich nach dem aktuellen Speichermodell und dem Vorhandensein von überschreibenden Zeiger-Modifizierern.

Bei Berechnungen mit Zeigern wird davon ausgegangen, daß der Zeiger auf ein Array von Objekten zeigt. So wird ein Zeiger, wenn er als Zeiger auf den Typ *type* deklariert ist, bei Hinzufügen eines Integerwertes um die Anzahl der Objekte vom Typ *type* vergrößert. Wenn *type* die Größe 10 hat und der Integerwert 5 zu einem Zeiger auf *type* hinzukommt, erhöht sich der Wert der Speicheradresse, die im Zeiger steht, um 50 (Byte). Der Wert der Differenz ist die Anzahl der Array-Elemente, die die beiden Zeiger voneinander unterscheidet. Wenn *ptr1* zum Beispiel auf das dritte Element eines Array und *ptr2* auf des zehnte Element des Array zeigt, dann ist das Ergebnis von *ptr2 - ptr1* 7. Wenn ein Integerwert zu einem »Zeiger auf den Typ *type*« addiert oder von ihm subtrahiert wird, ist das Ergebnis ebenfalls vom Typ »Zeiger auf den Typ *type*«.

Es gibt keinen »Zeiger auf das Element nach dem letzten Element«, aber ein Zeiger kann einen derartigen Wert annehmen. Wenn *P* auf das letzte Array-Element zeigt, ist *P + 1* legal, aber *P + 2* undefiniert. Wenn *P* auf das Element nach dem letzten Array-Element zeigt, ist *P - 1* legal, da es ein Zeiger auf das letzte Element ist. Die Anwendung des Indirektionsoperators * auf einen »Zeiger auf das Element nach dem letzten Element« führt zu undefiniertem Verhalten.

Sie können sich *P + n* als Inkrementierung des Zeigers um (*n* * **sizeof**(*type*)) Bytes vorstellen, solange der Zeiger innerhalb des zulässigen Bereichs bleibt (vom ersten Element bis zu dem Element nach dem letzten Element).

Wenn zwei Zeiger auf Elemente desselben Array subtrahiert werden, erhält man einen Integerwert vom Typ *ptrdiff_t*, der in stddef.h definiert ist (einen Wert

vom Typ **signed long** für **huge**- und **far**-Zeiger, einen Wert vom Typ **singed int** für alle anderen Zeiger). Dieser Wert repräsentiert die Differenz zwischen den Indizes von zwei in Frage kommenden Elementen, vorausgesetzt, sie liegt im Bereich von *ptrdiff_t*. Im Ausdruck *P1 - P2*, wobei *P1* und *P2* Zeiger auf den Typ *type* sind (oder Zeiger auf den qualifizierten Typ *type*, müssen *P1* und *P2* auf existierende Elemente zeigen oder auf das Element nach dem letzten Element. Wenn *P1* auf das *i*-te und P2 auf das *j*-te Element zeigt, hat *P1 - P2* den Wert (*i - j*).

Zeiger-konvertierungen

Zeigertypen können mit dem Mechanismus der »Typumwandlung« in andere Zeigertypen konvertiert werden:

```
char *str;
int *ip;
str = (char *) ip;
```

Allgemeiner ausgedrückt heißt dies: (*type**) konvertiert einen Zeiger in den Typ »Zeiger auf *type*«.

C++ Referenz-deklarationen

C++ Referenztypen sind eng mit den Zeigertypen verwandt. *Referenztypen* erzeugen Stellvertreter für Objekte und erlauben die Übergabe von Argumenten an Funktionen durch Referenz (*call by reference*). Im traditionellen C erfolgt die Argumentübergabe nur durch den *Wert* (*call by value*). In C++ können Sie Argumente durch Wert und Referenz übergeben. Im Abschnitt »Referenzieren« weiter hinten in diesem Kapitel finden Sie eine vollständige Beschreibung dieser Möglichkeit.

Arrays

Die Deklaration

type *Deklarator* [<*Konstantenausdruck*>]

deklariert ein Array, das sich aus Elementen des Typs *type* zusammensetzt. In C besteht ein Array aus einem kontinuierlichen Speicherbereich, der genauso groß ist, daß alle Elemente darin Platz finden.

Wenn in einem Array-Deklarator ein Ausdruck vorhanden ist, muß er eine positive Integerkonstante ergeben. Der Wert ist die Anzahl der Array-Elemente. Jedes Element eines Array ist numeriert und zwar von 0 bis zur Anzahl der Elemente minus 1.

Mehrdimensionale Arrays werden durch Deklaration von Arrays des Typs Array aufgebaut. Ein zweidimensionales Array namens *alpha*, mit fünf Zeilen und sieben Spalten, wird zum Beispiel folgendermaßen deklariert:

```
type alpha [5] [7];
```

Unter Umständen hat der erste von mehreren Array-Deklaratoren keinen Ausdruck in den Klammern. Ein derartiges Array ist von unbestimmter Größe. Dies ist dann zulässig, wenn die Größe eines Array nicht zur Speicherplatzreservierung erforderlich ist. Die genaue Größe des Array ist z.B. weder in der **extern**-Deklaration eines Array noch bei der Angabe eines Array-Funktionsparameters erforderlich. Eine Erweiterung in Turbo C++ (gegenüber ANSI C) ist die Möglichkeit, ein Array mit unbestimmter Größe als letztes Element in einer Struktur einzusetzen. Ein derartiges Array vergrößert nicht den Umfang der Struktur, es sei denn, es wird ein Dummy-Byte eingefügt, damit das Array an einer Wortgrenze beginnt. Diesen Strukturen wird normalerweise dynamisch Speicher zugewiesen. Die Größe des aktuellen Array muß explizit der Strukturgröße hinzugefügt werden, damit genügend Speicherplatz reserviert wird.

Ein Array-Typ-Ausdruck wird in einen konstanten Zeiger auf das erste Element des Array konvertiert. Dies ist jedoch nicht der Fall, wenn er der Operand eines **sizeof** oder **&**-Operators ist.

Funktionen

Funktionen sind zentrale Bestandteile von Turbo C++. Während bei Sprachen wie Pascal zwischen Prozeduren und Funktionen unterschieden wird, übernehmen in Turbo C++ die Funktionen beide Aufgabenbereiche.

Deklarationen und Definitionen

Jedes Programm muß eine externe Funktion mit dem Namen **main** haben, die den Beginn des Programms markiert. Funktionen werden normalerweise als Prototypen entweder in Standard- oder benutzerdefinierten Include-Dateien deklariert. Sie können auch innerhalb von Programmdateien deklariert werden. Funktionen sind per Voreinstellung vom Typ **extern**. Auf sie kann von jeder Programmdatei aus zugegriffen werden, aber der Zugriff kann mit dem Speicherklassen-Spezifizierer **static** eingeschränkt werden (vergleichen Sie den Abschnitt »Bindung« weiter vorne in diesem Kapitel).

Funktionen sind in den Quelltextdateien definiert oder werden durch Linken vorcompilierter Bibliotheken zur Verfügung gestellt.

Eine Funktion kann mehrere Male in einem Programm deklariert werden, vorausgesetzt, daß die Deklarationen kompatibel sind. Eine nicht-definierende Funktionsdeklaration, die im Prototypformat angegegeben wird, liefert Turbo C++ detaillierte Parameterinformationen. Dadurch kann die Anzahl der Argumente kontrolliert und die Zulässigkeit der Typen überprüft, sowie Typkonvertierung vorgenommen werden.

Hinweis: In C++ müssen immer Funktionsprototypen verwendet werden. Wir raten Ihnen, diese Prototypen auch in C einzusetzen.

Mit Ausnahme des Überladens von Funktionen in C++ ist nur eine Definition pro Funktion zugelassen. Eventuell vorhandene Deklarationen müssen mit dieser Definition übereinstimmen. (Der wichtigste Unterschied zwischen einer Definition und einer Deklaration besteht darin, daß die Definition einen Funktionsrumpf hat.)

Deklarationen und Prototypen

Bei Kernighan und Ritchie kann eine Funktion implizit durch ihr Vorhandensein in einem Funktionsaufruf oder wie folgt explizit deklariert werden:

<type>func()

wobei *type* der optionale Rückgabewert mit der Voreinstellung **int** ist. In C++ bedeutet obige Deklaration *<type>func*(*void*).

Eine Funktion kann so deklariert werden, daß sie einen beliebigen Typ mit Ausnahme eines Array oder einer Funktion liefert. Bei diesem Konzept kann der Compiler nicht überprüfen, ob der Typ oder die Anzahl der Argumente in dem Funtionsaufruf mit der Deklaration übereinstimmen. Dieses Problem wurde durch die Einführung von Funktionsprototypen mit folgender Deklarationssyntax behoben:

<type>func(*Parameter-Deklaratorliste*);

Deklaratoren legen den Typ für jeden Funktionsparameter fest. Der Compiler verwendet diese Information, um den Funktionsaufruf auf seine Gültigkeit hin zu überprüfen. Der Compiler kann dann auch Argumente in den benötigten Typ konvertieren.

Hinweis: Sie können den Compiler in der integrierten Entwicklungsumgebung oder in der Kommandozeilen-Version veranlassen, gegebenenfalls die Warnung »Function called without a prototype« auszugeben.

Nehmen wir an, Sie haben folgendes Programm-Fragment:

```
extern long lmax(long v1, v2);     /* Prototyp */

foo()
{
    int limit = 32;
    char ch = 'A';

    long mval;

    mval = lmax(limit,ch);  /* Funktionsaufruf */
}
```

Da das Programm für *lmax* einen Funktionsprototyp hat, werden *limit* und *ch* – gemäß den gebräuchlichen Konvertierungsregeln – in den Typ **long** konvertiert, bevor sie für den Aufruf von *lmax* auf dem Stack abgelegt werden.

Ohne Funktionsprototyp würden *limit* und *ch* als Integer und Zeichen auf den Stack gelegt. In diesem Fall würden die vom Stack an *lmax* übergebenen Werte in Größe und Inhalt nicht der von *lmax* erwarteten Übergabe entsprechen – und dies führt zu Problemen. Die klassische Deklaration läßt keine Überprüfung des Parametertyps und der Parameteranzahl zu. Deshalb ist der Einsatz von Funktionsprototypen eine große Hilfe bei der Fehlersuche.

Funktionsprototypen unterstützen auch die Programmdokumentation. Die Funktion *strcpy* zum Beispiel benötigt zwei Argumente: einen Quellstring und einen Zielstring. Die Frage ist, welcher ist welcher? Der Funktionsprototyp

```
char *strcpy(char *dest, char *source);
```

beantwortet diese Frage. Sie können sich eine Include-Datei, die Prototypen enthält, ausdrucken lassen. Damit steht Ihnen der größte Teil der Informationen zur Verfügung, die Sie in Ihrem Programm für den Aufruf dieser Funktionen benötigen. Wenn ein Prototyp-Parameter einen Bezeichner enthält, wird dieser nur für spätere Fehlermeldungen benutzt, die diesen Parameter betreffen; ansonsten wird er nicht verwendet.

Ein Funktionsdeklarator, der nur das Wort **void** in Klammern umfaßt, repräsentiert eine Funktion, die keine Argumente benötigt:

func(*void*);

In C++ bedeutet die Funktionsdeklaration *func*() dasselbe.

Ein Funktionsprototyp deklariert normalerweise eine Funktion mit einer festen Anzahl von Parametern. Bei Funktionen, die eine unterschiedliche Anzahl von Parametern haben können (wie zum Beispiel **printf**), kann der Funktionsprototyp mit einer Ellipse enden, zum Beispiel:

```
f(int *Zaehler, long Summe, ...)
```

Bei dieser Prototyp-Art werden die festen Parameter während des Compilierens überprüft und die variablen Parameter ohne Typüberprüfung weitergegeben.

Hinweis: stdarg.h enthält Makros, die Sie in selbstdefinierten Funktionen mit variabler Parameteranzahl verwenden können.

Im folgenden finden Sie einige Beispiele für Deklaratoren und Prototypen:

```
int f();                    /* In C eine Funktion ohne
                               Informationen über Parameter,
                               die einen int zurückliefert. Dies
                               ist klassischer K&R-Stil */
int f();                    /* In C++ eine Funktion ohne
                               Argumente */
int f(void);                /* Eine Funktion ohne Parameter,
                               die int liefert */
int p(int,long);            /* Eine Funktion mit zwei Para-
                               metern, int und long, die einen
                               int liefert */
int pascal q(void);         /* Eine Pascal-Funktion ohne
                               Parameter, die einen int liefert */
```

```
char far *s(char *source,
int kind);                /* Eine Funktion, die einen far-
                             Zeiger auf char liefert und zwei
                             Parameter akzeptiert: einen
                             Zeiger auf char und einen int. */
int printf(char
*format, ...);            /* Eine Funktion, die int liefert,
                             einen Zeiger auf einen festen
                             char-Parameter und weitere
                             Parameter von unbekanntem Typ
                             akzeptiert */
int (*fp) (int);          /* Ein Zeiger auf eine Funktion, die int
                             liefert und einen int-Parameter
                             akzeptiert */
```

Definitionen

In der folgenden Tabelle finden Sie die allgemeine Syntax für Funktionsdefinitionen.

Tabelle 2.9 Definition externer Funktionen

Datei:
Extern-Definition
Datei Extern-Definition

Extern-Definition:
Funktionsdefinition
Deklaration
asm-Anweisung

Funktionsdefinition:
<Mehrere-Deklarations-Spezifizierer>Deklarator <Deklarationsliste>
Blockanweisung

Eine Funktionsdefinition hat im allgemeinen die im folgenden aufgelisteten Bestandteile, wobei die Grammatik für kompliziertere Fälle weitere Komponenten zuläßt:

1. Optionale Speicherklassen-Spezifizierer: **extern** oder **static**; voreingestellt ist **extern**;

2. ein Rückgabetyp, auch **void** möglich; voreingestellt ist **int**.

Die Elemente von 1. und 2. können kombiniert werden.

3. Optionale Modifizierer: **pascal**, **cdecl**, **interrupt**, **near**, **far**, **huge**, **_export**, **_loadds**, **_saveregs**. Die Voreinstellung hängt vom Speichermodell und den Compilereinstellungen ab.

4. Name der Funktion;

5. Eine Parameter-Deklarationsliste (auch eine leere Liste möglich), die in runden Klammern steht. Wenn Sie in C eine leere Liste angeben möchten, sollten Sie **func**(*void*) schreiben. Die traditionelle Schreibweise **func**() ist zulässig, aber veraltet und möglicherweise unsicher.

6. Ein Funktionsrumpf, der den auszuführenden Programmcode für einen Funktionsaufruf repräsentiert.

Deklaration formaler Parameter

Die Syntax der Deklarationsliste für formale Parameter ist derjenigen der Deklaratoren in normalen Bezeichnerdeklarationen ähnlich.

Im folgenden finden Sie einige Beispiele:

```
int func(void)  {                    // keine Argumente

int func(T1 t1, T2 t2, T3 t3=1) {
                                     // (in C++) drei einfache
                                     // Parameter, der letzte mit
                                     // Vorgabewert

int func(T1* ptr1, T2& tref) {
                                     // (in C++) ein Zeiger- und
                                     // ein Referenzargument

int func(register int i) {           // fordert ein Register für
                                     // das Argument an

int func(char *str,...) {            // ein Stringargument und eine
                                     // variable Anzahl weiterer
                                     // Argumente oder eine feste
                                     // Anzahl von Argumenten mit
                                     // variablen Typen
```

In C++ können Sie einem formalen Parameter, wie oben aufgeführt, einen Vorgabewert zuordnen. Parameter mit voreingestellten Werten müssen die letzten Argumente in der Parameterliste sein. Die Argumente können vom Typ *Skalar*, *Struktur*, *Variante* oder *Aufzählung* sein, es kann sich auch um Zeiger oder Referenzen auf Strukturen und Varianten oder um Zeiger auf Funktionen oder Klassen handeln.

Die Ellipse (...) zeigt an, daß die Funktion bei verschiedenen Gelegenheiten mit unterschiedlich vielen Argumenten aufgerufen wird. Vor der Ellipse kann eine Liste mit bekannten Argumentdeklarationen stehen. Diese Form des Prototyps reduziert den Umfang der Überprüfung, die der Compiler durchführen kann. Die deklarierten Parameter haben während der Funktionsdauer einen automatischen Gültigkeitsbereich und automatische Dauer. Der einzige zulässige Speicherklassen-Spezifizierer ist **register**. Die Modifizierer **const** und **volatile** können verwendet werden.

Funktionsaufrufe und Argumentkonvertierungen

Beim Aufruf einer Funktion werden die aktuellen Argumente in der Reihenfolge ihrer formalen Parameter aufgeführt. Die Argumente werden – wie bei der Initialisierung – in die Typen der formalen Parameter konvertiert.

Im folgenden fassen wir die Regeln zusammen, die festlegen, wie Turbo C++ mit und ohne Prototypen die Sprachmodifizierer und formalen Parameter in Funktionsaufrufen behandelt.

1. Die Sprachmodifizierer für eine Funktionsdefinition müssen mit den Modifizierern in der Funktionsdeklaration bei allen Aufrufen der Funktion übereinstimmen.

2. Eine Funktion kann die Werte ihrer formalen Parameter ändern. Dies hat aber keine Auswirkung auf die Argumente in der aufrufenden Routine. Die Ausnahme bilden Referenzargumente in C++.

Wurde kein Funktionsprototyp deklariert, konvertiert Turbo C++ Integerargumente in einem Funktionsaufruf entsprechend den Integererweiterungsregeln (vergleichen Sie hierzu den Abschnitt »Standardkonvertierungen« weiter vorne in diesem Kapitel). Steht im Gültigkeitsbereich ein Prototyp, konvertiert Turbo C++ das gegebene Argument in den Typ des deklarierten Parameters, genauso wie bei einer Zuweisung.

Enthält ein Funktionsprototyp eine Ellipse (...), konvertiert Turbo C++ alle vorhandenen Funktionsargumente – bis hin zu der Ellipse – genauso wie bei jedem anderen Prototyp. Der Compiler konvertiert alle weiteren Argumente entsprechend den normalen Regeln für Funktionsargumente ohne Prototypen.

Ist ein Prototyp vorhanden, muß die Anzahl der Argumente mit der der Parameter übereinstimmen (außer es ist eine Ellipse vorhanden). Die Typen müssen nur soweit kompatibel sein, daß sie – wie bei einer Zuweisung – konvertiert werden können. Sie können immer eine explizite Typenumwandlung (cast) zur Konvertierung eines Arguments in einen für den Funktionsprototyp zulässigen Typ verwenden.

Wichtig: Stimmt der Funktionsprototyp nicht mit der Funktionsdefinition überein, findet Turbo C++ dies nur dann heraus, wenn die Definition in derselben Übersetzungseinheit wie der Prototyp steht. Wenn Sie eine Bibliothek für Routinen mit einer entsprechenden Prototyp-Include-Datei anlegen, müssen Sie darauf achten, daß beim Compilieren der Bibliothek die Include-Datei mit einbezogen wird, damit Diskrepanzen zwischen den Protoypen und den Definitionen entdeckt werden. C++ verfügt über eine typensichere Bindung, so daß Unterschiede zwischen erwarteten und tatsächlichen Parametern durch den Linker ausgeschlossen werden.

Strukturen

Eine *Struktur* ist ein abgeleiteter Typ, der im Normalfall für eine vom Benutzer definierte Ansammlung von benannten Elementen (oder Komponenten) steht.

Die Elemente können mit wenigen Einschränkungen, die später erläutert werden, jeden (standardmäßigen oder abgeleiteten) Typ annehmen und in beliebiger Reihenfolge aufgeführt werden. Außerdem kann in einer Struktur auch ein Bitfeld verwendet werden, was sonst nicht zulässig ist. Mit dem Turbo C++ Struktur-Typ lassen sich komplexe Datenstrukturen fast genauso leicht handhaben wie einzelne Variablen.

In C++ wird ein Strukturtyp wie ein Klassentyp behandelt, allerdings gibt es gewisse Unterschiede: Der voreingestellte Zugriff und die Voreinstellung für die Basisklasse sind beide **public**. Die C++ Zugriffs-Spezifizierer **public**, **private** und **protected** erlauben eine bessere Kontrolle des Zugriffs auf Strukturelemente. Abgesehen von dieser optionalen Zugangskontrolle und den erwähnten Ausnahmen gilt die folgende Beschreibung der Struktursyntax und -benutzung gleichermaßen für C und C++. Strukturen werden durch das Schlüsselwort **struct** definiert. Zum Beispiel:

```
struct mystruct { ... };  // mystruct ist der "Tag"-Name der Struktur
...
struct mystruct s, *ps, arrs[10];
/* s  ist vom Typ struct mystruct;
   ps ist vom Typ Zeiger auf struct mystruct;
   arrs ist ein Array von struct mystruct. */
```

Strukturen und Typdefinitionen ohne »Tag«-Name

Wenn man den »Tag«-Namen der Struktur wegläßt, erhält man eine *unbenannte* Struktur. Sie können unbenannte Strukturen nutzen, um die durch Komma voneinander getrennten Bezeichner in der **struct**-Bezeichnerliste mit dem angegebenen (oder von ihm abgeleiteten) Strukturtyp zu deklarieren. Diese unbenannten Strukturtypen können Sie nicht nochmals zur Deklaration von Objekten an einer anderen Stelle verwenden:

```
struct { ... } s, *ps, arrs[10];  //unbenannte Struktur
```

Beim Deklarieren einer Struktur können Sie ein **typedef** mit oder ohne »Tag« erzeugen.

```
typedef struct mystruct { ... } MYSTRUCT;
MYSTRUCT s, *ps, arrs[10];
                                  // dasselbe wie struct mystruct s,
                                  // usw.
typedef struct { ... } YRSTRUCT;  // kein "Tag"
YRSTRUCT y, *yp, arry[20];
```

Normalerweise verwenden Sie entweder einen »Tag«-Namen oder ein **typedef** mit Strukturtypnamen. Beide können in weiteren Deklarationen verwendet werden.

Hinweis: Unbenannte Struktur- und Variantenelemente werden bei der Initialisierung ignoriert.

Strukturelement-Deklarationen

Die *Element-Deklarationsliste* innerhalb der Strukturklammern deklariert die Typen und Namen der Strukturelemente, wie sie in Tabelle 2.2 weiter vorne in diesem Kapitel zusammengefaßt werden. Ein Element kann bis auf zwei Ausnahmen jeden Typ annehmen:

1. Der Typ eines Elements kann nicht mit dem Typ der Struktur identisch sein, die gerade deklariert wird (in C++ kann das Schlüsselwort **struct** weggelassen werden):

```
struct mystruct { mystruct s } s1, s2;    // falsch
```

Ein Element kann jedoch ein Zeiger auf die zu deklarierende Struktur sein:

```
struct mystruct { mystruct *ps } s1, s2;  // in Ordnung
```

Ebenso kann eine Struktur schon definierte andere Strukturen enthalten.

2. Außer in C++ kann ein Element nicht den Typ »Funktion, die ... liefert« annehmen, sondern nur den Typ »Zeiger auf Funktion ... «. In C++ kann eine Struktur Funktionen als Elemente enthalten.

Strukturen und Funktionen

Eine Funktion kann sowohl eine Struktur als auch einen Zeiger auf eine Struktur zurückgeben.

```
mystruct func1(void);       // func1() liefert eine Struktur zurück
mystruct *func2(void);      // func2() liefert einen Zeiger
                            // auf eine Struktur zurück
```

Eine Struktur kann folgendermaßen an eine Funktion übergeben werden:

```
void func1(mystruct s);     // direkt
void func2(mystruct *sptr); // über einen Zeiger
void func3(mystruct &sref); // als Referenz (nur in C++)
```

Zugriff auf die Strukturelemente

Auf Struktur- und Variantenelemente kann mit den Operatoren . und -> zugegriffen werden. Nehmen wir an, daß das Objekt *s* vom Strukturtyp *S* und *sptr* ein Zeiger auf *S* ist. Wenn dann weiterhin *m* ein in *S* deklarierter Elementbezeichner von Typ *M* ist, sind die Ausdrücke *s.m* und *sptr->m* vom Typ *M* und repräsentieren beide das Elementobjekt *m* in *s*. Der Ausdruck *sptr->m* ist gleichbedeutend (aber einfacher) mit (**sptr*).*m*.

Der Operator . ist der direkte und -> der indirekte Zugang zu Strukturelementen. Sehen Sie hierzu folgendes Beispiel:

```
struct mystruct {
    int i;
    char str[21];
    double d;
} s, *sptr=&s;
...
s.i = 3;              // Zuweisung an das Element i von mystruct s
sptr->d = 1.23;       // Zuweisung an das Element d von mystruct s
```

Der Ausdruck *s.m* ist ein L-Wert (Linkswert) unter der Bedingung, daß *s* kein L-Wert und *m* kein Array ist. Der Ausdruck *sptr->m* ist ein L-Wert, außer, *m* ist ein Array. Wenn die Struktur *B* die Struktur *A* enthält, dann kann auf die Elemente von A auf verschiedene Weise zugegriffen werden.

```
struct A {
   int j;
   double x;
};

struct B {
   int i;
   struct A a;
   double d;
} s, *sptr;
...
s.i = 3;              // Zuweisung auf das Element i von B
s.a.j = 2;            // Zuweisung auf das Element j von A
sptr->d = 1.23;       // Zuweisung auf das Element d von B
(sptr->a).x = 3.14;   // Zuweisung auf das Element       x von A
```

Jede Strukturdeklaration führt einen eigenen Strukturtyp ein. So sind durch die Definitionen

```
struct A {
    int i,j;
    double d;
} a, a1;

struct B {
    int i,j;
    double d;
} b;
```

beide Objekte *a* und *a1* Strukturen vom Typ *A*, aber die Typen der Objekte *a* und *b* sind verschieden. Strukturen können einander nur dann zugewiesen werden, wenn Quelle und Ziel denselben Typ haben.

```
a = a1;               // in Ordnung: derselbe Typ,
                      // daher Element-für-Element-Zuweisung
```

```
a = b;                              // falsch: verschiedene Typen
a.i = b.i; a.j = b.j; a.d = b.d
                                    // aber Sie können die Elemente
                                    // einzeln zuweisen
```

Ausrichtung auf Wortgrenzen in Strukturen

Speicher wird einer Struktur Element für Element von links nach rechts und von den niedrigen Adressen zu den höheren hin zugewiesen. In dem folgenden Beispiel:

```
struct mystruct {
    int i;
    char str[21];

    double d;
} s;
```

belegt das Objekt *s* genügend Speicher für einen 2-Byte-Integer, einen 21-Byte-String und einen 8-Byte-**double**. Die Anordnung dieses Objekts im Speicher wird durch die Turbo C++ Option *Ausrichtung an geraden Adressen* bestimmt. Ist sie ausgeschaltet (dies ist die Voreinstellung), belegt *s* 31 Bytes ohne Lücken. Wird sie im Dialogfenster *Optionen für Code-Generierung* eingeschaltet, füllt Turbo C++ die Struktur so mit einzelnen Bytes auf, daß

1. die Struktur an einer Wortgrenze startet (gerade Adresse);

2. jedes Element, das nicht vom Typ **char** ist, auf einer geraden Adresse relativ zum Beginn der Struktur zu liegen kommt;

3. die Struktur eine insgesamt gerade Anzahl von Bytes umfaßt (nötigenfalls durch Anhängen eines einzelnen Bytes).

Wäre *Word alignment* eingeschaltet, so wäre die obige Struktur 32 Bytes lang, vor dem **double** würde ein Byte eingeschoben.

Namensbereich von Strukturen

»Tag«-Namen von Strukturen, Varianten und Aufzählungstypen teilen sich denselben Namensbereich (bis auf Aufzählungstypen innerhalb von Strukturen, die in C++ einen anderen Namensbereich haben). Das bedeutet, daß derartige »Tags« innerhalb eines Gültigkeitsbereichs eindeutig sein müssen. Sie müssen sich aber nicht von Bezeichnern in den anderen drei Namensbereichen unterscheiden: dem Label-, Element- und Einzel-Namensbereich (der aus Variablen, Funktionen, **typedef**-Namen und Aufzählungstypen besteht).

Elementnamen innerhalb einer Struktur müssen eindeutig sein, können sich aber in anderen Strukturen wiederholen:

```
goto s;
...
s:
struct s  {   // richtig: "Tag"- und Label-Namensbereiche
              // sind verschieden
   int s;     // richtig: Label-, "Tag"- und Element-Namens-
              // bereiche sind verschieden
   float s;   // falsch: Elementname doppelt
} s;          // richtig: Variablen-Namensbereich ist unter-
              // schiedlich. In C++ ist dies nur möglich,
              // wenn s keinen Konstruktor hat.
union s  {    // falsch: Gleicher Struktur- und Variantenname
              // im "Tag"-Namensbereich
   int s;     // richtig: Neuer Element-Namensbereich
   float f;
} f;          // richtig: Variablen-Namensbereich

struct t {
   int s;     // richtig: Neuer Element-Namensbereich
   ...
} s;          // falsch: Variablenname doppelt
```

Unvollständige Deklarationen

Ein Zeiger auf eine Struktur vom Typ *A* kann in der Deklaration einer anderen Struktur *B* bereits verwendet werden, bevor die Struktur *A* deklariert ist:

```
struct A;                 // unvollständig
struct B { struct A *pa };
struct A { struct B *pb };
```

Die erste Nennung von *A* ist *unvollständig*, weil *A* an diesem Punkt noch nicht definiert wird. Unvollständige Deklarationen sind in diesem Zusammenhang aber erlaubt, weil für die Definition von *B* die Größe von *A* nicht benötigt wird.

Bitfelder

Eine Struktur kann jede Kombination von Bitfeld- und Nicht-Bitfeld-Elementen enthalten. Sie können **signed** und **unsigned** Integerelemente als Bitfelder mit einer Größe von 1 bis 16 definieren. Sie spezifizieren die Größe des Bitfeldes und den optionalen Bezeichner wie folgt:

Typ-Spezifizierer <Bitfeld-Bezeichner>: Größe;

Als *Typ-Spezifizierer* sind **char**, **unsigned char**, **int** oder **unsigned int** erlaubt. Bitfelder werden innerhalb eines Wortes vom niederwertigen zum höherwertigen Bit hin angeordnet. Die *Größe* eines Bitfeldes muß angegeben werden und einen konstanten Wert zwischen 0 und 16 haben.

Wenn man den Bezeichner des Bitfeldes wegläßt, wird die in *Größe* angegebene Größe des Bitfeldes angelegt, aber das Feld ist nicht verwendbar. Damit können z.B. Hardware-Register nachgebildet werden, in denen einige Bits unbenutzt sind. Das folgende Beispiel hat das in Abbildung 2.1 dargestellte Layout zur Folge.

```
struct mystruct {
    int      i : 2;
    unsigned j : 5;
    int        : 4;
    int      k : 1;
    unsigned m : 4;
} a, b, c:
```

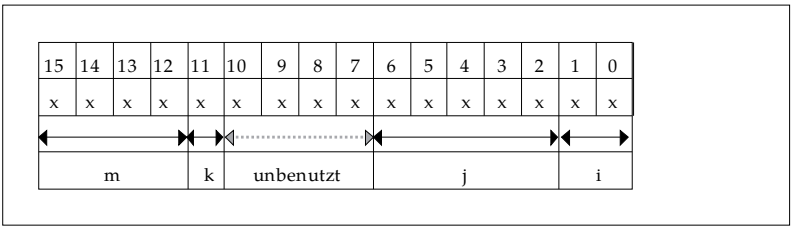


Abbildung 2.1 Das resultierende Layout

Integerfelder werden in Zweier-Komplementform angelegt, wobei das linke Bit das höchstwertige ist. Wird das Integerfeld als **signed int** deklariert, so wird das linke Bit als Vorzeichenbit interpretiert. Ein Bitfeld der Größe 2, in dem der Binärwert 11 gespeichert ist, würde deswegen als 3 interpretiert, wenn es als **unsigned** und als -1, wenn es als **int** deklariert ist. Im vorhergehenden Beispiel würde die erlaubte Zuweisung `a.i = 6`den Binärwert 10, der dezimal -2 bedeutet, in *a.i* ohne Warnung hinterlassen. Das als **signed int** deklarierte Bitfeld *k* mit der Größe 1 kann dann natürlich nur die Werte -1 und 0 annehmen, weil der Bitwert 1 als -1 interpretiert wird.

Hinweis: Bitfelder können nur in Strukturen, Varianten und Klassen verwendet werden. Auf Bitfelder wird genauso zugegriffen wie auf alle anderen Elemente (. und ->). Allerdings erzeugen Bitfelder einige Probleme, wenn portierbarer Code geschrieben werden soll, da die Organisation von Bits innerhalb von Bytes und von Bytes innerhalb von Wörtern maschinenabhängig ist.

Der Ausdruck *&mystruct.x* ist nicht erlaubt, wenn x ein Bitfeld ist, da es keine Garantie dafür gibt, daß *mystruct.x* auf einer Byte-Adresse liegt.

Varianten

C++ Varianten (unions) sind das Gegenstück zum varianten Recordtyp von Pascal und Modula-2. Die Variante ist ein abgeleiteter Typ, der in vielen syntaktischen und funktionalen Eigenschaften mit der Struktur übereinstimmt. Der hauptsächliche Unterschied zur Struktur ist, daß in der Variante zu einem

gegebenen Zeitpunkt nur eines der Elemente aktiv sein kann. Die Größe einer Variante wird von der Größe seines größten Elements bestimmt. Es kann jeweils nur der Wert eines der Elemente gespeichert werden. Im folgenden Beispiel

```
union myunion { // Varianten-"Tag" = myunion
   int i;
   double d;
   char ch;
} mu, *muptr=&mu;
```

kann der Bezeichner *mu*, der vom Typ **union** *myunion* ist, dazu benutzt werden, einen 2-Byte-Integer, einen 8-Byte-**double** oder einen Einzel-Byte-**char** zu speichern (aber nur jeweils einen dieser drei Typen).

sizeof(*union myunion*) und **sizeof**(*mu*) geben beide 8 zurück. Allerdings sind 6 Bytes davon ungenutzt, wenn *mu* verwendet wird, um einen Integer zu speichern bzw. 7 Bytes, wenn *mu* verwendet wird, um einen **char** zu speichern. Auf die Elemente einer Variante wird genauso zugegriffen wie auf die Elemente einer Struktur (. und ->).

Aber Vorsicht:

```
mu.d = 4.016;
printf ("mu.d = %  f\n",mu.d);   // richtig: zeigt mu.d = 4.016
printf ("mu.i = %  d\n",mu.i);   // unbrauchbares Resultat
mu.ch = 'A';
printf ("mu.ch = %c\n",mu.ch); // richtig: zeigt mu.ch = A
printf ("mu.d = %  f\n",mu.d);   // unbrauchbares Resultat
mptr->i = 3;
printf ("mu.i = %  d\n",mu.i);   // richtig: zeigt mu.i = 3
```

In diesem Beispiel ist das zweite *printf* zwar erlaubt, weil *mu.i* wirklich ein Integertyp ist. Allerdings entspricht das Bitmuster in *mu.i* einem Teil des vorher zugewiesenen **double,** und das dürfte wohl kaum ein brauchbares Resultat ergeben.

Bei einer korrekten Konvertierung zeigt ein Zeiger auf eine Variante auf jedes Element der Variante und umgekehrt.

Anonyme Varianten (C++)

Eine Variante, die keinen Namen hat und nicht zur Deklaration eines benannten Objekts (oder anderen Typs) verwendet wird, heißt *anonyme Variante* und hat folgende Syntax:

union { *Elementliste* };

Auf ihre Elemente kann in dem Gültigkeitsbereich, in dem die Variante deklariert ist, direkt – ohne die *x.y* oder *p->y*-Syntax – zugegriffen werden.

Anonyme Varianten können keine Elementfunktionen und enthalten und müssen auf Dateiebene als statisch definiert sein. Mit anderen Worten: eine anonyme Variante kann keine externe Bindung haben.

Deklarationen von Varianten

Die Deklarationssyntax für Varianten stimmt fast vollständig mit der für Strukturen überein. Es gibt folgende Unterschiede:

1. Varianten können Bitfelder enthalten, aber nur jeweils eines der Bitfelder kann aktiv sein. Alle diese Bitfelder beginnen am Anfang der Variante (beachten Sie, daß Bitfelder maschinenabhängig sind und es dadurch beim Schreiben von portierbarem Programmtext zu Problemen kommen kann).

2. Anders als C++ Strukturen können C++ Varianten nicht die Klassenzugriff-Spezifizierer **public**, **private** und **protected** verwenden. Sämtliche Felder einer Variante sind **public**.

3. Varianten können nur über ihr erstes Element initialisiert werden:

```
union local87 {
   int i;
   double d;
} a = { 20 };
```

4. In C++ kann eine Variante kein Bestandteil einer Klassenhierarchie sein. Sie kann weder eine Basisklasse sein noch von einer anderen Klasse abgeleitet werden. Eine Variante kann aber einen Konstruktor haben.

Aufzählungstypen

Der Aufzählungstyp erlaubt es, statt einer Menge von Integerwerten entsprechende Bezeichner zu verwenden. In folgendem Beispiel wird eine Variable *anyday* vom Aufzählungstyp (**enum**) *days* deklariert und dazu eine Menge von Aufzählungselementen (*sun*, *mon*, etc), die für konstante Integerwerte stehen.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
```

Turbo C++ speichert Aufzählungstypen in einzelnen Bytes, wenn *Enum als Integer behandeln* im Menü *Optionen/Compiler/Code-Generierung* ausgeschaltet ist. Per Voreinstellung ist diese Option eingeschaltet (d.h. Aufzählungstypen sind also normalerweise vom Typ **int**), wenn der Wertebereich es zuläßt. In Ausdrücken werden die Werte von Aufzählungstypen immer auf Integer ausgedehnt. Die Bezeichner eines Aufzählungstyps sind vom Typ **signed char**, **unsigned char** oder **int**, abhängig vom Wertebereich der Aufzählungstypen. Können alle Werte in einem **signed** oder **unsigned char** repräsentiert werden, wird dieser Typ auch verwendet.

In C kann einer Aufzählungstypvariablen jeder Wert vom Typ **int** zugewiesen werden – es gibt keine darüberhinausgehende Typenprüfung. In C++ dagegen kann einer derartigen Variable nur einer der aufgezählten Bezeichner zugewiesen werden.

Zum Beispiel:

```
anyday = mon;   // richtig
anyday = 1;     // falsch, selbst wenn mon == 1 ist
```

Der optionale »Tag«-Name *days* kann in folgenden Deklarationen von Aufzählungsvariablen des Typs **enum days** verwendet werden.

```
enum days payday, holiday;   //deklariert zwei Variablen
```

In C++ können Sie das Schlüsselwort **enum** auch weglassen, wenn *days* im aktuellen Gültigkeitsbereich keine andere Bedeutung hat. Genauso wie bei Strukturen und Varianten kann man den »Tag«-Namen weglassen, wenn keine weiteren Variablen dieses Aufzählungstyps benötigt werden:

```
enum { sun, mon, tues, wed, thur, fri, sat } anyday;
/* anonymer Aufzählungstyp */
```

Aufzählungsbezeichner innerhalb geschweifter Klammern heißen *Aufzählungstyp-Konstanten* (vgl. den entsprechenden Abschnitt in Kapitel 1 »Lexikalische Grammatik«). Jeder dieser Konstanten ist ein fester Integerwert zugewiesen. Sind keine expliziten Initialisierer vorhanden, wird die erste Konstante (*sun*) auf 0 gesetzt und die folgenden in Einerschritten aufsteigend belegt (*mon* = 1, *tues* = 2, etc.).

Bei Vorhandensein von expliziten Initialisierern können Sie aber eine oder mehrere Konstanten auf einen bestimmten Wert setzen. Alle folgenden Konstanten werden dann mit jeweils um 1 steigenden Werten initialisiert. In dem Beispiel

```
// Der Initialisierungsausdruck kann Aufzählungskonstanten ent-
// halten, die vorher deklariert wurden
enum coins { penny = 1, tuppence, nickel = penny + 4, dime = 10,
             quarter = nickel * nickel } smallchange;
```

hat *tuppence* den Wert 2, *nickel* den Wert 5 und *quarter* den Wert 25. Zur Initialisierung kann jeder Ausdruck verwendet werden, der einen positiven oder negativen Wert ergibt (nach möglichen Integer-Ausdehnungen). Diese Werte sind normalerweise eindeutig, doppelte Werte sind erlaubt.

Aufzählungstypen können immer dann verwendet werden, wenn **int**-Typen zulässig sind.

```
enum days { sun, mon, tues, wed, thur, fri, sat } anyday;
enum days payday;
typedef enum days DAYS;
DAYS *daysptr;
int i = tues;
anyday = mon;         // richtig
*daysptr = anyday;    // richtig
mon = tues;           // falsch: mon ist eine Konstante
```

Die »Tag«-Namen von Aufzählungstypen teilen sich den Namensbereich mit Strukturen und Varianten. Die aufgezählten Konstanten teilen sich den Namensbereich mit normalen Variablen.

```
int mon = 11;

{
    enum days {sun, mon, tues, wed, thur, fri, sat } anyday;
     // Die Aufzählungskonstante mon verdeckt die außerhalb
     // deklarierte Variable int mon
    struct days { int i, j;}; // falsch: doppelter "Tag" days
    double sat;               // falsch: Neudefinition von sat
}

mon = 12;                     // hier gilt wieder int mon
```

In C++ gelten Aufzählungstyp-Konstanten, die in einer Klasse definiert wurden, nur im Gültigkeitsbereich dieser Klasse.

Ausdrücke

Ein *Ausdruck* ist eine Folge von Operatoren, Operanden und Interpunktionszeichen, die eine Berechnung definiert.

Die formale Syntax in Tabelle 2.11 zeigt, daß Ausdrücke rekursiv definiert sind. Für die Schachtelungstiefe gibt es keine formale Begrenzung. (Allerdings wird der Compiler die Fehlermeldung `out of memory` bringen, wenn er einen zu komplexen Ausdruck übersetzen soll). Aus Tabelle 2.11 ersehen Sie, wie Bezeichner und Operatoren zu grammatikalisch korrekten Ausdrücken kombiniert werden können.

Die Auswertung von Ausdrücken richtet sich nach bestimmten Konvertierungs- und Gruppierungsregeln, nach der Abarbeitungsreihenfolge und -richtung der Operatoren, dem Vorhandensein von Klammern und den Datentypen der Operanden. Die Reihenfolge, in der Sie Operanden und Unterausdrücke anordnen, muß nicht notwendigerweise die Reihenfolge sein, in der Turbo C++ Ihre Ausdrücke auswertet (vgl. den Abschnitt »Auswertungsreihenfolge« weiter hinten in diesem Kapitel). Die Standardkonvertierungen sind detailliert in Tabelle 2.6 beschrieben. Ausdrücke können einen L-Wert, einen R-Wert oder keinen Wert ergeben. Ausdrücke können zu Nebeneffekten führen, gleichgültig, ob sie einen Wert ergeben oder nicht.

In Tabelle 2.11 sehen Sie die komplette Definition der Abarbeitungsreihenfolge und -richtung (Assoziativität) der Operatoren. Wir haben diese Informationen noch einmal in Tabelle 2.10 zusammengefaßt.

Es gibt 16 Kategorien für die Abarbeitungsreihenfolge, von denen manche nur einen Operator enthalten. Operatoren derselben Kategorie werden in dieser Beziehung gleichbehandelt. Operatoren, die zweimal in der Tabelle enthalten sind, sind beim ersten Auftreten unär und beim zweiten Auftreten binär.

Jede Kategorie hat ihre eigene Abarbeitungsrichtung: von links nach rechts oder von rechts nach links. Sind keine Klammern vorhanden, werden Ausdrücke nach Operatoren derselben Kategorie zerlegt.

Die Reihenfolge in dieser Tabelle entspricht der Rangfolge der einzelnen Kategorien. Die erste Kategorie (die erste Zeile) hat den höchsten Rang.

Tabelle 2.10 Rangfolge und Assoziativität von Turbo C++ Operatoren

Operator	Assoziativität
`()  []  ->  ::  .`	Links nach rechts
`!  ~  +  -  ++  --  &(typecast)` **sizeof new delete**	Rechts nach links
`.*  ->*`	Links nach rechts
`*  /  %`	Links nach rechts
`+  -`	Links nach rechts
`<<  >>`	Links nach rechts
`<  <=  >  >=`	Links nach rechts
`==  !=`	Links nach rechts
`&`	Links nach rechts
`^`	Links nach rechts
`|`	Links nach rechts
`&&`	Links nach rechts
`||`	Links nach rechts
`?:` (bedingter Ausdruck)	Rechts nach links
`=  *=  /=  =%  +=  -=  &=  ^=  |=  <<=  >>=`	Rechts nach links
`,`	Links nach rechts

Tabelle 2.11 Turbo C++ Ausdrücke

Primärer-Ausdruck:
- *Literal*
- **this** (C++)
- `::` *Bezeichner* (C++)
- `::` *Operator-Funktionsname* (C++)
- `::` *Qualifizierter-Name*
- (*Ausdruck*)
- *Name*

Literal:
- *Integerkonstante*
- *Zeichenkonstante*
- *Gleitkommakonstante*
- *String-Konstante*

Name:
- *Bezeichner*
- *Operator-Funktionsname* (C++)
- *Konvertierungs-Funktionsname* (C++)
- *~ Klassenname*
- *Qualifizierter-Name* (C++)

Qualifizierter-Name: (C++)
- *Qualifizierter-Klassenname* `::` *Name*

Suffix-Ausdruck:
- *Primärer-Ausdruck*

Suffix-Ausdruck [*Ausdruck*]
Suffix-Ausdruck (*<Ausdrucksliste>*)
Einfacher-Typname (*<Ausdrucksliste>*) (C++)
Suffix-Ausdruck . *Name*
Suffix-Ausdruck -> *Name*
Suffix-Ausdruck ++
Suffix-Ausdruck --

Ausdrucksliste:
Zuweisungsausdruck
Ausdrucksliste , *Zuweisungsausdruck*

Unärer-Ausdruck:
Suffix-Ausdruck
++ *Unärer-Ausdruck*
-- *Unärer-Ausdruck*
Unärer-Operator cast-Ausdruck
sizeof *Unärer-Ausdruck*
sizeof (*Typname*)
Reservierungsausdruck (C++)
Freigabeausdruck (C++)

Unärer-Operator: einer der folgenden
&*+-~!

Reservierungsausdruck: (C++)
<::>new <Plazierung>Eingeschränkter-Typname <Initialisierer>
<::>new <Plazierung> (*Typname*) *<Initialisierer>*

Plazierung: (C++)
(*Ausdrucksliste*)

Eingeschränkter-Typname: (C++)
Typ-Spezifizierer<Eingeschränkter-Deklarator>

Eingeschränkter-Deklarator: (C++)
Zeiger-Operator<Eingeschränkter-Deklarator>
Eingeschränkter-Deklarator [*<Ausdruck>*]

Freigabeausdruck: (C++)
*<::>***delete** *cast-Ausdruck*
*<::>***delete** [] *cast-Ausdruck*

cast-Ausdruck:
Unärer-Ausdruck
(*Typname*) *cast-Ausdruck*

pm-Ausdruck:
cast-Ausdruck
pm-Ausdruck .* *cast-Ausdruck* (C++)
pm-Ausdruck ->* *cast-Ausdruck* (C++)

Multiplikativer-Ausdruck:
pm-Ausdruck

Multiplikativer-Ausdruck * *pm-Ausdruck*
Multiplikativer-Ausdruck / *pm-Ausdruck*
Multiplikativer-Ausdruck % *pm-Ausdruck*

Additiver-Ausdruck:
Multiplikativer-Ausdruck
Additiver-Ausdruck + *Multiplikativer-Ausdruck*
Additiver-Ausdruck - *Multiplikativer-Ausdruck*

Schiebeoperations-Ausdruck:
Additiver-Ausdruck
Schiebeoperations-Ausdruck << *Additiver-Ausdruck*
Schiebeoperations-Ausdruck >> *Additiver-Ausdruck*

Relationaler-Ausdruck:
Schiebeoperations-Ausdruck
Relationaler-Ausdruck < *Schiebeoperations-Ausdruck*
Relationaler-Ausdruck > *Schiebeoperations-Ausdruck*
Relationaler-Ausdruck <= *Schiebeoperations-Ausdruck*
Relationaler-Ausdruck >= *Schiebeoperations-Ausdruck*

Gleichheitsausdruck
Relationaler-Ausdruck
Gleichheitsausdruck == *Relationaler-Ausdruck*
Gleichheitsausdruck != *Relationaler-Ausdruck*

UND-Ausdruck:
Gleichheitsausdruck
UND-Ausdruck & *Gleichheitsausdruck*

Exklusives-ODER-Ausdruck:
UND-Ausdruck
Exklusives-ODER-Ausdruck ^ *UND-Ausdruck*

Inklusives-ODER-Ausdruck:
Exklusives-ODER-Ausdruck
Inklusives-ODER-Ausdruck | *Exklusives-ODER-Ausdruck*

Logisches-UND-Ausdruck:
Inklusives-ODER-Ausdruck
Logisches-UND-Ausdruck && *Inklusives-ODER-Ausdruck*

Logisches-ODER-Ausdruck:
Logisches-UND-Ausdruck
Logisches-ODER-Ausdruck || *Logischer-UND-Ausdruck*

Bedingter-Ausdruck:
Logisches-ODER-Ausdruck
Logisches-ODER-Ausdruck ? *Ausdruck* : *Bedingter-Ausdruck*

Zuweisungsausdruck:
Bedingter-Ausdruck
Unärer-Ausdruck Zuweisungsoperator Zuweisungsausdruck

Zuweisungsoperator: einer der folgenden:
=*=/=%=+=-=
<<=>>=&=^=|=

Ausdruck:
Zuweisungsausdruck
Ausdruck , *Zuweisungsausdruck*

Konstantenausdruck:
Bedingter-Ausdruck

Ausdrücke in C++

C++ erlaubt das Überladen von einigen Standardoperatoren (vgl. »Überladene Operatoren« in Kapitel 3 »C++«). Ein überladener Operator verhält sich in spezieller vordefinierter Art und Weise, wenn er in Ausdrücken eines Klassentyps verwendet wird. So könnte zum Beispiel der relationale Operator == in der Klasse *complex* so definiert werden, daß er zwei komplexe Zahlen auf Gleichheit überprüft, ohne daß seine normale Verwendbarkeit in Datentypen, die keiner Klasse angehören, berührt wird. Ein überladener Operator ist als Funktion implementiert; diese Funktion bestimmt den Typ der Operanden, den L-Wert und die Auswertungsreihenfolge, die angewendet werden, wenn ein überladener Operator benutzt wird. Das Überladen kann die Abarbeitungsreihenfolge eines Operators allerdings nicht ändern. C++ erlaubt auch benutzerdefinierte Konvertierungen zwischen Klassenobjekten und grundlegenden Typen.

Beachten Sie, daß einige der Regeln, die in diesem Kapitel behandelt werden, nicht unbedingt für Ausdrücke in C++ zutreffen.

Auswertungsreihenfolge

Die Reihenfolge, in der Turbo C++ den Operanden eines Ausdrucks Werte zuweist, ist nicht festgelegt, ausgenommen, ein Operator legt eine bestimmte Reihenfolge fest. Der Compiler wird versuchen, den Ausdruck so anzuordnen, daß ein möglichst effizienter Code entsteht. Vorsicht ist daher in Ausdrücken angebracht, in denen ein Wert mehr als einmal modifiziert wird. Generell sollte man die Verwendung von Ausdrücken vermeiden, die Werte eines Objektes sowohl ändern als auch benützen. Betrachten Sie den folgenden Ausdruck:

```
i = v[i++];    // i ist undefiniert
```

Der Wert von *i* hängt davon ab, ob *i* vor oder nach der Zuweisung erhöht wird. Genauso ist

```
int total = 0;
sum = (total = 3) + (++total);   // sum = 4 oder sum = 7 ??
```

für *sum* und *total* mehrdeutig.

Die Lösung ist, eine temporäre Variable zu verwenden:

```
int temp, total = 0;
temp = ++total;
sum = (total = 3) + temp;
```

Wo die Syntax allerdings eine Auswertungsreihenfolge erzwingt, kann man risikolos mehrere Auswertungen einsetzen:

```
sum = (i = 3, i++, i++);  // richtig: sum = 4, i = 5
```

Jeder Unterausdruck des durch Komma getrennten Ausdrucks in Klammern wird von links nach rechts ausgewertet und der gesamte Ausdruck nimmt den Wert ganz rechts an (der hier erst nach der Auswertung hochgezählt wird).

Turbo C++ gruppiert Ausdrücke um, indem assoziative und kommutative Operatoren ohne Berücksichtigung von Klammern so umgestellt werden, daß eine möglichst hohe Effizienz erreicht wird. In keinem Fall beeinflußt die Umgruppierung den Wert des Ausdrucks.

Sie können Klammern benutzen, um eine bestimmte Abarbeitungsreihenfolge zu erzwingen. Wenn Sie zum Beispiel die Variablen *a*, *b*, *c* und *f* haben, dann erzwingt der Ausdruck $f = a + (b+c)$, daß $(b+c)$ einen Wert erhält, bevor *a* dazugezählt wird.

Fehler und Überläufe

Während der Auswertung eines Ausdrucks können viele problematische Situationen auftreten, etwa eine Division durch 0 oder ein zu großer Gleitkommawert. Der Überlauf von Integervariablen wird ignoriert (C verwendet für *n*-Bit-Register Modulo-2^n-Arithmetik). Fehler, die von den Gleitkomma-Routinen erkannt werden, können aber mit Standard- oder benutzerdefinierten Routinen behandelt werden. Lesen Sie dazu die Erläuterungen zu *matherr* und *signal* in der Online-Hilfe.

Operatoren-Semantik

Die hier beschriebenen Turbo C++ Operatoren stimmen mit den Standard ANSI-C-Operatoren überein.

Falls die Operatoren nicht überladen werden, treffen die folgenden Erläuterungen für C und C++ zu. In C++ können alle Operatoren überladen werden, mit Ausnahme von `.` (Elementoperator) und `?:` (Bedingungsoperator). (Ebenso können die C++ Operatoren `::` und `.*` nicht überladen werden.)

Die folgenden Ausführungen treffen für überladene Operatoren möglicherweise nicht zu. Tabelle 2.11 beschreibt die Syntax aller Operatoren und Ausdrücke mit Operatoren.

Beschreibung der Operatoren

Operatoren sind Symbole, die Berechnungen auslösen, wenn sie auf Variable und andere Objekte eines Ausdrucks angewendet werden. Turbo C++ verfügt über sehr viele Operatoren – nicht nur über die »normalen« arithmetischen und logischen, sondern auch über Operatoren für Manipulationen auf der Bit-Ebene, für den Zugriff auf Struktur- und Variantenkomponenten sowie für Zeigeroperationen (Referenzierung und Dereferenzierung).

In C++ gibt es weitere Operatoren für den Zugriff auf Klassenelemente und deren Objekte sowie einen Mechanismus für überladene Operatoren. *Überladen* bedeutet, daß Sie die Aktion von Standardoperatoren neu definieren können, wenn diese auf die Objekte einer Klasse angewendet werden.

Im folgenden Abschnitt beschränken wir uns auf die Darstellung der Standardoperatoren in Turbo C++. Nach der Vorstellung dieser Operatoren folgt eine Erläuterung der Datentypen und Deklarationen und deren Auswirkungen auf die Aktionen der einzelnen Operatoren. Danach stellen wir Ihnen die Syntax für den Aufbau von Ausdrücken aus Operatoren, Interpunktionszeichen und Objekten vor. In C++ stehen Ihnen folgende Operatoren zur Verfügung:

Operator: eines der folgenden Zeichen

```
[]().->++--&*+
-~!                          sizeof   /%<<>><
><=>===!=^|&&||
?:=*=/=%=+=-=<<=>>=
&=^=|=,###
```

Die Operatoren # und ## werden nur vom Präprozessor verwendet.

Folgende Operatoren sind nur in C++ verfügbar:

```
::.*->*
```

Mit Ausnahme von [], () und ?:, die Ausdrücke klammern, werden Operatoren, die sich aus mehreren Zeichen zusammensetzen, als ein einziges Symbol betrachtet. Ein und derselbe Operator kann je nach Kontext unterschiedlich interpretiert werden, zum Beispiel:

`A * B`	Multiplikation
`*ptr`	Dereferenzierung (Indirektion)
`A & B`	Bitweiser UND-Operator
`&A`	Adressierungsoperation
`int &`	Referenz-Modifizierer (nur C++)
`label:`	Anweisungs-Label
`a ? x : y`	Bedingte Anweisung
`void func(int n);`	Funktionsdeklaration
`a = (b+c) *d;`	Geklammerter Ausdruck
`a, b, c;`	Komma-Ausdruck
`func(a, b, c);`	Funktionsaufruf
`a = ~b;`	Bitweise Negation
`~func() {delete a;}`	Destruktor (nur C++)

Unäre Operatoren

&	Adreßoperator
*	Indirektionsoperator
+	Unärer Plus-Operator
-	Unärer Minus-Operator
~	Bit-Komplement-Operator
!	Logischer Negationsoperator
++	Präfix-, Suffix-Inkrement-Operator
--	Präfix-, Suffix-Dekrement-Operator

Binäre Operatoren

Additive Operatoren

+	Binärer Plus-Operator (Addition)
-	Binärer Minus-Operator (Subtraktion)

Multiplikative Operatoren

*	Multiplikation
/	Division
%	Prozent

Schiebeoperatoren

<<	Linksschiebeoperator
>>	Rechtsschiebeoperator

Bitweise Operatoren

&	Bitweiser UND-Operator
^	Bitweise Exklusives-ODER-Operator
|	Bitweise Inklusives-ODER-Operator

Logische Operatoren

&&	Logischer UND-Operator
||	Logischer ODER-Operator

Zuweisungs-Operatoren

=	Einfache Zuweisung
*=	Zuweisung – Produkt
/=	Zuweisung – Quotient
%=	Zuweisung – Modulo
+=	Zuweisung – Summe

-= Zuweisung – Differenz
<<= Zuweisung – Linksschiebeoperator
>>= Zuweisung – Rechtsschiebeoperator
&= Zuweisung – Bitweises-UND-Operator
^= Zuweisung – Bitweises-Exklusives-ODER-Operator
!= Zuweisung – Bitweises-ODER-Operator

Relationale Operatoren

< Kleiner-als-Operator
> Größer-als-Operator
<= Kleiner-gleich-Operator
>= Größer-gleich-Operator

Gleichheitsoperatoren

== Gleich-Operator
!= Ungleich-Operator

Komponenten-Auswahl-Operatoren

. Direkter Komponenten-Auswahl-Operator
-> Indirekter Komponenten-Auswahl-Operator

Klassenelement-Operatoren

:: Gültigkeitsbereich-Zugriffsoperator
.* Dereferenzierungszeiger auf Klassenelemente
->* Dereferenzierungszeiger auf Klassenelemente

Bedingungsoperator

a ? x :y "wenn a, dann x; andernfalls y"

Komma-Operator

, Auswertung, z. B. `a, b, c;` von links nach rechts

Suffix- und Präfix-Operatoren

Die sechs Suffix-Operatoren `[ ] ( ) . -> ++` und `--` werden benutzt, um Suffix-Ausdrücke zu bilden (wie aus der Syntax-Tabelle 2.11 hervorgeht). Inkrement- und Dekrement-Operator (`++` und `--`) sind zudem auch Präfixe und unäre Operatoren. Sie werden im Abschnitt »Unäre Operatoren« weiter hinten in diesem Kapitel erläutert.

Array-Indizierungsoperator [ ]
Seine Syntax lautet:

Suffix-Ausdruck [*Ausdruck*]

Im obigen Ausdruck muß entweder *Suffix-Ausdruck* oder *Ausdruck* ein Zeiger und die andere Komponente dann ein Integerwert sein.

In C, jedoch nicht immer in C++, ist der Ausdruck *exp1*[*exp2*] definiert als

```
* ((exp1) + (exp2))
```

wobei *exp1* ein Zeiger und *exp2* ein Integer oder *exp1* ein Integer und *exp2* ein Zeiger ist. (Die Interpunktionszeichen [ ], * und + können in C++ überladen werden.)

Funktionsaufruf-Operator ()
Der Ausdruck

Suffix-Ausdruck (<*Argument-Ausdrucksliste*>)

ist der Aufruf einer Funktion, die vom Suffix-Ausdruck bestimmt wird. Die *Argument-Ausdrucksliste* ist eine durch Kommas getrennte Liste von Ausdrücken jeglichen Typs, die die Funktionsargumente darstellen. Der Wert des Funktionsaufruf-Ausdrucks wird, falls vorhanden, von der **return**-Anweisung in der Funktionsdefinition bestimmt. Näheres dazu finden Sie im Abschnitt »Funktionsaufrufe und Argumentkonvertierungen« weiter vorne.

Struktur-/Varianten-Elementoperator .
In dem Ausdruck

Suffix-Ausdruck . *Bezeichner*

muß der Suffix-Ausdruck eine Struktur oder eine Variante und der Bezeichner der Name eines der Elemente dieser Struktur oder Variante sein. Der Ausdruck wählt dieses Element an. Der Wert des Ausdrucks ist gleich dem Wert des ausgewählten Elements. Es handelt sich nur dann um einen L-Wert, wenn der Suffix-Ausdruck ein L-Wert ist. Beispiele für die Benutzung von . und -> in Strukturen finden Sie im Abschnitt »Zugriff auf Strukturelemente«.

Struktur-/Varianten-Zeigeroperator ->
In dem Ausdruck

Suffix-Ausdruck -> *Bezeichner*

muß der Suffix-Ausdruck ein Zeiger auf eine Struktur oder Variante und der Bezeichner der Name eines Elements dieser Struktur oder Variante sein. Der Ausdruck wählt dieses Element an. Der Wert des Ausdrucks ist gleich dem Wert des ausgewählten Elements. Es handelt sich nur dann um einen L-Wert, wenn der Suffix-Ausdruck ein L-Wert ist.

Suffix-Inkrement-Operator ++
In dem Ausdruck

Suffix-Ausdruck ++

ist der Suffix-Ausdruck der Operand; er muß einem skalaren Typ (arithmetischen oder Zeigertyp) angehören und ein modifizierbarer L-Wert sein. Der Wert des Ausdrucks ist gleich dem Wert des Suffix-Ausdrucks *vor* der Inkrementie-

rung. Nachdem der Suffix-Ausdruck ausgewertet ist, wird der Operand um 1 inkrementiert.

Der (interne) Inkrementwert ist vom Typ des Operanden abhängig. Zeigertypen werden gemäß den Regeln der Zeigerarithmetik erhöht.

Suffix-Dekrement-Operator
Der *Suffix-Dekrement-Operator* hat folgende Syntax:

Suffix-Ausdruck `--`

Für den Suffix-Dekrement-Operator gelten dieselben Regeln wie für den Suffix-Inkrement-Operator, mit der Ausnahme, daß der Operand nach der Auswertung um 1 *dekrementiert* wird.

Inkrement- und Dekrement-Operatoren

Die ersten beiden der nun folgenden unären Operatoren sind `++` und `--`. Sie sind sowohl Suffix- als auch Präfix-Operatoren. Aus diesem Grund werden sie gesondert erläutert. Die übrigen sechs unären Operatoren werden im Anschluß an diesen Abschnitt beschrieben.

Präfix-Inkrement-Operator
In dem Ausdruck

`++` *Unärer-Ausdruck*

ist der unäre Ausdruck der Operand; er muß einen skalaren Typ haben und ein modifizierbarer L-Wert sein. Der Operand wird um 1 inkrementiert, *bevor* der Ausdruck zugewiesen wird. Der Wert des gesamten Ausdrucks ist der inkrementierte Wert des Operanden. Der (interne) Inkrementwert ist wiederum vom Typ des Operanden abhängig. Zeigertypen gehorchen den Regeln der Zeigerarithmetik.

Präfix-Dekrement-Operator
Der Präfix-Dekrement-Operator hat folgende Syntax:

`--` *Unärer Ausdruck*

Er folgt denselben Regeln wie der Präfix-Inkrement-Operator, mit der Ausnahme, daß der Wert des Operanden vor der Auswertung um 1 *dekrementiert* wird.

Unäre Operatoren

Die sechs unären Operatoren (neben `++` und `--`) sind `&` `*` `+` `-` `~` und `!`. Ihre Syntax lautet:

Unärer-Operator cast-Ausdruck

cast-Ausdruck:
Unärer-Ausdruck
(*Typname*) *cast-Ausdruck*

Der Adreßoperator &
Die Operatoren & und * (der Operator * wird im nächsten Abschnitt erläutert) sind *Referenz-* bzw. *Dereferenz*-Operatoren. In dem Ausdruck

& *cast-Ausdruck*

muß der *cast-Ausdruck* entweder eine Funktion oder ein L-Wert sein, der kein Bitfeld darstellt und nicht als **register** deklariert ist. Ist der Operand vom Typ *type*, dann ist der Wert des Ausdrucks ein Zeiger auf *type*.

Das Symbol *&* wird in C++ auch für die Spezifizierung von Referenztypen verwendet (siehe Kapitel 3 »C++«).

Hinweis: Einige Ausdrücke, die keine L-Werte sind, wie zum Beispiel Funktionen und Arrays, werden in bestimmten Zusammenhängen automatisch in Zeigertypen konvertiert. Der Operator & kann auch bei derartigen Objekten verwendet werden, was aber überflüssig wäre und darum vermieden werden sollte.

Sehen Sie sich die folgenden Zeilen an:

```
type t1 = 1, t2 = 2;

type *ptr = &t1;      // initialisierter Zeiger

*ptr = t2;            // dieselbe Bedeutung wie t1 = t2
```

Beachten Sie, daß *type *ptr =&t1* genauso behandelt wird wie

```
T *ptr;

ptr = &t1;
```

das heißt, daß *ptr* und nicht **ptr* initialisiert wird. Nach der Initialisierung von *ptr* mit der Adresse *&t1* kann die Dereferenzierung zum L-Wert **ptr* stattfinden.

Der Indirektionsoperator *
In dem Ausdruck

* *cast-Ausdruck*

muß der *cast-Ausdruck* vom Typ »Zeiger auf *type*« sein, wobei *type* jeder Typ sein kann. Das Ergebnis der Indirektion ist vom Typ *type*. Ist der Operand ein Zeiger auf eine Funktion, dann ist das Ergebnis der Funktionsrückgabewert; ist der Operand ein Zeiger auf ein Objekt, so ist das Ergebnis der L-Wert des Objekts.

In diesen Situationen ist das Ergebnis der Indirektion nicht definiert:

1. Der *cast-Ausdruck* ist ein Null-Zeiger.

2. Der *cast-Ausdruck* ist die Adresse einer automatischen Variablen und die Ausführung des Anweisungsblocks, in dem sie definiert ist, ist schon beendet.

Unärer Plus-Operator +
In dem Ausdruck

+ *cast-Ausdruck*

muß der *cast-Ausdruck* ein arithmetischer Typ sein. Das Ergebnis ist der Wert des Ausdrucks nach dessen Auswertung.

Unärer Minus-Operator -
In dem Ausdruck

- *cast-Ausdruck*

muß der *cast-Ausdruck* ein arithmetischer Typ sein. Das Ergebnis ist der mit -1 multiplizierte Wert des Ausdrucks nach dessen Auswertung.

Bit-Komplement-Operator ~
In dem Ausdruck

~ *cast-Ausdruck*

muß der *cast-Ausdruck* ein Integertyp sein. Das Ergebnis ist das bitweise Komplement des Operanden nach dessen Auswertung. Jedes 0-Bit des Operanden wird auf 1 gesetzt und jedes 1-Bit auf 0.

Logischer Negationsoperator !
In dem Ausdruck

! *cast-Ausdruck*

muß der *cast-Ausdruck* ein skalarer Typ sein. Das Ergebnis ist vom Typ **int** und die logische Negation des Operanden, also 0, wenn der Operand ungleich 0 ist und 1, wenn der Operand gleich 0 ist. Der Ausdruck *!E* ist gleichwertig zu dem Ausdruck (*0 == E*).

Der sizeof-Operator

Es gibt zwei Anwendungen für den **sizeof**-Operator:

sizeof *Unärer-Ausdruck*
sizeof (*Typname*)

Das Ergebnis ist in beiden Fällen eine Integerkonstante, die den vom Operanden belegten Platz in Bytes angibt (bis auf wenige Ausnahmen vom Typ bestimmt). Im ersten Anwendungsfall wird der Typ des Operandenausdrucks ohne dessen Auswertung bestimmt (und hat deswegen keine Nebeneffekte). Wenn der Operand vom Typ **char** (**signed** oder **unsigned**) ist, ergibt **sizeof** den Wert 1. Wenn der Operand ein Array ist und nicht aus einer Parameterübergabe stammt, zeigt das Ergebnis die Anzahl aller Bytes des Array an (das heißt, ein Array-Name wird hier *nicht* in einen Zeiger konvertiert). Die Anzahl der Elemente eines Array ist gleich dem Ausdruck **sizeof** *array*/**sizeof** *array*[*0*].

Ist der Operand ein Übergabeparameter vom Array- oder Funktionstyp, gibt **sizeof** die Größe des Zeigers an. Wird **sizeof** auf Strukturen oder Varianten angewendet, erhält man die Summe der Bytes, einschließlich der Auffüllungen.

sizeof kann nicht im Zusammenhang mit Ausdrücken vom Typ Funktion, unvollständigen Typen, geklammerten Ausdrücken mit diesen Typen oder mit L-Werten, die für ein Bitfeld stehen, verwendet werden.

Der Typ des Ergebnisses von **sizeof** ist *size_t*, das in der Datei stddef.h als **unsigned int** definiert ist. Sie können **sizeof** in Präprozessor-Direktiven verwenden (nur in Turbo C++).

In C++ liefert **sizeof**(*Klassentyp*) die Größe eines Objekts dieses Typs. Hat *Klassentyp* einen Vorfahren, ist die Größe des Vorfahren im Ergebnis enthalten.

Hinweis: Der Umfang des Speicherplatzes für die einzelnen Variablentypen hängt von dem verwendeten Rechner ab.

Multiplikative Operatoren

Es gibt drei multiplikative Operatoren: * / und %, deren Syntax folgendermaßen aussieht:

Multiplikativer-Ausdruck:
cast-Ausdruck
Multiplikativer-Ausdruck * *cast-Ausdruck*
Multiplikativer-Ausdruck / *cast-Ausdruck*
Multiplikativer-Ausdruck % *cast-Ausdruck*

Die Operanden von * (Multiplikation) und / (Division) müssen einen arithmetischen Typ, die Operanden von % (Modulo) einen Integertyp haben. Auf die Operanden werden die normalen arithmetischen Konvertierungen angewendet (vergleichen Sie dazu den Abschnitt »Standardkonvertierungen« weiter vorne in diesem Kapitel).

Das Ergebnis von (*op1* * *op2*) ist das Produkt der beiden Operanden. Das Ergebnis von (*op1* / *op2*) ist der Quotient, das von (*op1* % *op2*) ist der Rest, der bei der Division von *op1* durch *op2* anfällt (*op2* darf nicht Null sein). Der Gebrauch von / oder % mit `0` als zweitem Operanden ergibt einen Fehler.

Wenn *op1* und *op2* Integerzahlen sind, aber der Quotient kein Integer ist, ergibt sich folgendes:

1. Wenn *op1* und *op2* dasselbe Vorzeichen haben, ergibt (*op1* / *op2*) den größten Integer, der kleiner ist als der wahre Quotient. (*op1* % *op2*) hat das Vorzeichen von *op1*.

2. Wenn *op1* und *op2* verschiedene Vorzeichen haben, ergibt sich für (*op1* / *op2*) der kleinste Integer, der größer als der wahre Quotient ist. (*op1* % *op2*) hat das Vorzeichen von *op1*.

Additive Operatoren

Es gibt zwei additive Operatoren, + und -, deren Syntax folgendermaßen aussieht:

Additiver-Ausdruck:
Multiplikativer-Ausdruck
Additiver-Ausdruck + *Multiplikativer-Ausdruck*
Additiver-Ausdruck - *Multiplikativer-Ausdruck*

Der Additionsoperator +
Für *op1* + *op2* sind folgende Operanden zulässig:

1. *op1* und *op2* sind arithmetische Typen.
2. *op1* ist ein Integertyp und *op2* ist ein Zeiger auf ein Objekt.
3. *op2* ist ein Integertyp und *op1* ist ein Zeiger auf ein Objekt.

Im Fall 1 sind die Operanden den normalen arithmetischen Konvertierungsregeln unterworfen, das Ergebnis ist die arithmetische Summe der Operanden. In den Fällen 2 und 3 finden die Regeln für die Zeigerarithmetik Anwendung. (Vergleichen Sie dazu den Abschnitt »Zeigerarithmetik« weiter vorne in diesem Kapitel.)

Der Subtraktionsoperator -
Für *op1* - *op2* sind folgende Operanden zulässig:

1. *op1* und *op2* sind arithmetische Typen.
2. *op1* und *op2* sind Zeiger auf miteinander kompatible Objekte. (*Hinweis:* der nicht-qualifizierte Typ *type* ist kompatibel mit den Typen **const** *type*, **volatile** *type* und **const volatile** *type*.)
3. *op1* ist ein Zeiger auf ein Objekt und *op2* ist ein Integertyp.

Im Fall 1 sind die Operanden den normalen arithmetischen Konvertierungsregeln unterworfen und das Ergebnis ist die arithmetische Differenz der Operanden. In den Fällen 2 und 3 gelten die Regeln der Zeigerarithmetik.

Bitweise Schiebeoperatoren

Es gibt zwei bitweise Schiebeoperatoren, << und >>, deren Syntax folgendermaßen lautet:

Schiebeoperationsausdruck-Ausdruck:
Additiver-Ausdruck
Schiebeoperationsausdruck-Ausdruck << *Additiver-Ausdruck*
Schiebeoperationsausdruck-Ausdruck >> *Additiver-Ausdruck*

Links- und Rechtsschiebeoperatoren << und >>

In den Ausdrücken *E1* << *E2* und *E1* >> *E2* müssen die Operanden *E1* und *E2* Integertypen sein. *E1* und *E2* werden in der üblichen Weise ausgedehnt und der Typ des Ergebnisses ist der ausgedehnte Typ von *E1*. Ist *E2* negativ oder größer oder gleich der Bitbreite von *E1*, so ist die Operation undefiniert.

Das Ergebnis von *E1* << *E2* ist der Wert von *E1* um *E2* Bitpositionen nach links verschoben und (falls erforderlich) von rechts mit Nullen aufgefüllt. Ist *E1* vom Typ **unsigned long**, entspricht das Verschieben nach links der Multiplikation von *E1* mit 2^{E2}, vermindert um modulo *ULONG_MAX* +1. Das Linksschieben eines **unsigned int** *E1* entspricht der Multiplikation von *E1* mit 2^{E2}, vermindert um modulo *UINT_MAX* +1. (Die Konstanten *ULONG_MAX* und *UINT_MAX* sind in der Datei limits.h definiert.) Ist *E1* ein **signed**-Integer, muß das Ergebnis mit Vorsicht betrachtet werden, da das Vorzeichen wechseln kann.

Das Ergebnis von *E1* >> *E2* ist der Wert von *E1* um *E2* Bitpositionen nach rechts verschoben. Ist *E1* vom Typ **unsigned**, so wird (falls erforderlich) von links mit Nullen aufgefüllt; ist *E1* **signed**, wird von links mit dem Vorzeichenbit aufgefüllt (0 bei positivem, 1 bei negativem *E1*).

Dadurch ist sichergestellt, daß *E1* >> *E2* und *E1* das gleiche Vorzeichen haben. Der Wert von *E1* >> *E2* ist der Integerteil des Quotienten $E1/2^{E2}$ (dies gilt jedoch nicht, wenn *E1* vom Typ **signed** ist).

Relationale Operatoren

Es gibt vier relationale Operatoren, < > <= und >=, deren Syntax folgendermaßen lautet:

Relationaler-Ausdruck:
Schiebeoperations-Ausdruck
Relationaler-Ausdruck < *Schiebeoperations-Ausdruck*
Relationaler-Ausdruck > *Schiebeoperations-Ausdruck*
Relationaler-Ausdruck <= *Schiebeoperations-Ausdruck*
Relationaler-Ausdruck >= *Schiebeoperations-Ausdruck*

Der Kleiner-als-Operator <

Im Ausdruck *E1* < *E2* müssen die Operanden einer der folgenden Bedingungen genügen:

1. Sowohl *E1* als auch *E2* sind arithmetische Typen.

2. Sowohl *E1* als auch *E2* sind Zeiger auf qualifizierte oder unqualifizierte Versionen kompatibler Objekttypen.

3. Sowohl *E1* als auch *E2* sind Zeiger auf qualifizierte oder unqualifizierte Versionen kompatibler unvollständiger Typen.

Im Fall 1 werden die normalen arithmetischen Konvertierungsregeln durchgeführt. Das Ergebnis von *E1* < *E2* ist vom Typ **int**. Ist der Wert von *E1* kleiner als der Wert von *E2*, so ist das Ergebnis 1 (wahr); andernfalls ist es 0 (falsch).

In den Fällen 2 und 3, bei denen *E1* und *E2* Zeiger auf kompatible Typen sind, hängt das Ergebnis von *E1* < *E2* von den relativen Adressen der beiden Objekte ab, auf die die Zeiger zeigen. Werden Elemente derselben Struktur verglichen, so bedeutet der »höhere« Zeiger eine spätere Deklaration. Innerhalb eines Array bedeutet der »höhere« Zeiger einen größeren Index. Alle Zeiger, die auf Elemente derselben Variante zeigen, werden als gleich erkannt.

Normalerweise ergibt der Vergleich von Zeigern auf verschiedene Strukturen, Arrays oder Varianten oder von Zeigern, die außerhalb eines Array liegen, undefinierte Resultate.

Eine Ausnahme besteht bei Zeigern, die hinter das letzte Element zeigen (vergleichen Sie dazu den Abschnitt »Zeigerarithmetik« weiter vorne in diesem Kapitel). Wenn *P* auf ein Element eines Array zeigt und *Q* auf dessen letztes Element, so ist der Ausdruck *P* < *Q* + *1* erlaubt und hat den Wert 1 (wahr), obwohl *Q* + *1* nicht auf ein Element des Array zeigt.

Der Größer-als-Operator >

Der Ausdruck *E1* > *E2* ergibt 1 (wahr), wenn der Wert von *E1* größer als der Wert von *E2* ist; ansonsten ist das Ergebnis 0 (falsch). Für den arithmetischen bzw. den Zeigervergleich werden dieselben Regeln angewendet wie bei dem Kleiner-als-Operator. Es gelten auch dieselben Operanden-Regeln.

Der Kleiner-gleich-Operator <=

Der Ausdruck *E1* <= *E2* ergibt 1 (wahr), wenn der Wert von *E1* kleiner oder gleich *E2* ist; ansonsten ergibt sich der Wert 0 (falsch). Es gelten dieselben Regeln wie bei dem Kleiner-als-Operator.

Der Größer-gleich-Operator >=

Der Ausdruck *E1* >= *E2* ergibt den Wert 1 (wahr), wenn der Wert von *E1* größer oder gleich *E2* ist; ansonsten ergibt sich der Wert 0 (falsch). Es gelten dieselben Regeln wie bei dem Kleiner-als-Operator.

Gleichheits-operatoren

Es gibt zwei Gleichheitsoperatoren: == und !=. Sie testen die Gleichheit bzw. Ungleichheit von arithmetischen Werten oder Zeigern.

Die Anwendungsregeln sind denen für die relationalen Operatoren sehr ähnlich. Beachten Sie jedoch, daß == und != eine niedrigere Abarbeitungspriorität haben als die relationalen Operatoren <, > <=und >=. Sie können bestimmte Zeigertypen miteinander vergleichen, bei denen die relationalen Operatoren nicht erlaubt sind.

Die Syntax für diese Operatoren sieht folgendermaßen aus:

Gleichheitsausdruck:
Relationaler Ausdruck
Gleichheitsausdruck == *Relationaler Ausdruck*
Gleichheitsausdruck != *Relationaler Ausdruck*

Der Gleich-Operator ==

In dem Ausdruck *E1* == *E2* müssen die Operanden eine der folgenden Bedingungen erfüllen:

1. *E1* und *E2* sind arithmetische Typen.

2. *E1* und *E2* sind Zeiger auf qualifizierte oder unqualifizierte Versionen kompatibler Typen.

3. Einer von beiden ist ein Zeiger auf ein Objekt oder einen unvollständigen Typ und der andere ist ein Zeiger auf eine qualifizierte oder unqualifizierte Version des Typs **void**.

4. Einer von beiden ist ein Zeiger und der andere die Null-Zeigerkonstante.

Wenn die Typen von *E1* und *E2* auch für die relationalen Operatoren erlaubt sind, gelten dieselben Vergleichsregeln wie dort.

Im Fall 1 werden die normalen arithmetischen Konvertierungen durchgeführt, und das Ergebnis von *E1* == *E2* ist vom Typ **int**. Ist der Wert von *E1* gleich dem Wert von *E2*, so ist das Ergebnis 1 (wahr), ansonsten ist das Ergebnis 0 (falsch).

Im Fall 2 ergibt sich 1 (wahr), wenn *E1* und *E2* auf dasselbe Objekt zeigen, wenn beide hinter das letzte Element desselben Array zeigen oder wenn beide Nullzeiger sind.

Wenn *E1* und *E2* Zeiger auf eine Funktion sind, ergibt *E1* == *E2* den Wert 1 (wahr), wenn sie beide Null-Zeiger oder Zeiger auf dieselbe Funktion sind. Ergibt umgekehrt *E1* == *E2* den Wert 1 (wahr), zeigen entweder *E1* und *E2* auf dieselbe Funktion oder sie sind beide Null. Im Fall 4 wird der Zeiger auf ein Objekt oder einen unvollständigen Typ in den Typ des anderen Operanden konvertiert (Zeiger auf eine qualifizierte oder unqualifizierte Version des Typs **void**).

Der Ungleich-Operator !=

Für den Ausdruck *E1* != *E2* gelten dieselben Regeln wie für *E1* == *E2*, außer daß das Ergebnis 1 (wahr) ist, wenn die Operanden verschieden sind und 0 (falsch), wenn die Operanden gleich sind.

Der bitweise UND-Operator &

Seine Syntax lautet:

UND-Ausdruck:
Gleichheitsausdruck
UND-Ausdruck & *Gleichheitsausdruck*

In dem Ausdruck *E1* & *E2* müssen beide Operanden Integertypen sein. Die normalen arithmetischen Konvertierungen werden durchgeführt und das Ergebnis ist das bitweise UND von *E1* und *E2*.

Jedes Bit im Ergebnis wird durch folgende Tabelle bestimmt:

Tabelle 2.12 Tabelle mit bitweisen Operatoren

Bit-Wert in E1	Bit-Wert in E2	E1 & E2	E1 ^ E2	E1 | E2
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Der bitweise Exklusives-ODER-Operator ^

Seine Syntax lautet:

Exklusives-ODER-Ausdruck:
UND-Ausdruck
Exklusives-ODER-Ausdruck ^ *UND-Ausdruck*

Im Ausdruck *E1* ^ *E2* müssen beide Operanden Integertypen sein.
Die normalen arithmetischen Konvertierungen werden durchgeführt und das Ergebnis ist das bitweise exklusive ODER von *E1* und *E2*.
Jedes Bit des Ergebnisses wird bestimmt, wie in Tabelle 2.12 aufgeführt.

Der bitweise Inklusives-ODER-Operator |

Seine Syntax lautet:

Inklusives-ODER-Ausdruck:
Exklusives-ODER-Ausdruck
Inklusives-ODER-Ausdruck | *Exklusives-ODER-Ausdruck*

In dem Ausdruck *E1* | *E2* müssen beide Operanden Integertypen sein.
Die normalen arithmetischen Konvertierungen werden durchgeführt und das Ergebnis ist das bitweise inklusive ODER von *E1* und *E2*.
Jedes Bit des Ergebnisses wird bestimmt, wie in Tabelle 2.12 aufgeführt.

Der logische UND-Operator &&

Seine Syntax lautet:

Logischer-UND-Ausdruck:
Inklusives-ODER-Ausdruck
Logischer-UND-Ausdruck && *Inklusives-ODER-Ausdruck*

In dem Ausdruck *E1 && E2* müssen beide Operanden skalare Typen sein.

Das Ergebnis ist vom Typ **int** und hat den Wert 1 (wahr), wenn sowohl *E1* als auch *E2* einen Wert ungleich 0 haben, andernfalls ist das Ergebnis 0 (falsch).

Anders als beim bitweisen UND-Operator &, werden bei && die Operanden von links nach rechts ausgewertet. *E1* wird zuerst ausgewertet.

Wenn *E1* nach der Auswertung 0 ist, so ergibt *E1 && E2* 0 (falsch), und E2 wird nicht ausgewertet.

Der logische ODER-Operator ||

Seine Syntax lautet:

Logischer-ODER-Ausdruck:
Logischer-UND-Ausdruck
Logischer-ODER-Ausdruck || *Logischer UND-Ausdruck*

In dem Ausdruck E1 || E2 müssen beide Operanden skalare Typen sein. Das Ergebnis ist vom Typ **int** und hat den Wert 1 (wahr), wenn wenigstens einer der Operanden *E1* oder *E2* einen Wert ungleich 0 hat, andernfalls ist das Ergebnis 0 (falsch).

Anders als beim bitweisen Operator | werden bei || die Operanden von links nach rechts ausgewertet. Ist *E1* danach ungleich 0, so ergibt *E1* || *E2* 1 (wahr) und *E2* wird nicht ausgewertet.

Der Bedingungs-operator ?:

Seine Syntax lautet:

Bedingter-Ausdruck:
Logischer-ODER-Ausdruck
Logischer-ODER-Ausdruck ? *Ausdruck* : *Bedingter-Ausdruck*

In dem Ausdruck *E1 ? E2 : E3* muß der Operand *E1* ein skalarer Typ sein. Die Operanden *E2* und *E3* müssen einer der folgenden Regeln entsprechen:

1. Beide Operanden sind arithmetische Typen.
2. Beide Operanden sind kompatible Strukturen oder Varianten.
3. Beide Operanden sind vom Typ **void**.
4. Beide Operanden sind Zeiger auf qualifizierte oder unqualifizierte Versionen zueinander kompatibler Typen.
5. Einer der Operanden ist ein Zeiger, der andere die Null-Zeigerkonstante.
6. Einer der beiden Operanden ist ein Zeiger auf ein Objekt oder auf einen unvollständigen Typ, der andere ist ein Zeiger auf eine qualifizierte oder unqualifizierte Version vom Typ **void**.

Zuerst wird *E1* ausgewertet; falls sein Wert ungleich 0 ist, wird *E2* ausgewertet und *E3* ignoriert. Falls *E1* 0 ist, wird *E3* ausgewertet und *E2* wird ignoriert. Das Ergebnis von *E1 ? E2 : E3* ist also entweder der Wert von *E2* oder von *E3*.

Hinweis: In C++ ist das Ergebnis ein L-Wert.

Im Fall 1 werden für *E2* und *E3* die normalen arithmetischen Konvertierungen durchgeführt, und der Typ des Ergebnisses entspricht dem des jeweiligen Operanden.

Im Fall 2 entspricht der Typ des Ergebnisses dem Typ der Struktur oder Varianten von *E2* und *E3*.

Im Fall 3 ist das Ergebnis vom Typ **void**.

In den Fällen 4 und 5 ist das Ergebnis vom Typ Zeiger auf einen Typ, der alle Qualifizierer der Typen enthält, auf die die beiden Operanden zeigen.

Im Fall 6 entspricht der Typ des Ergebnisses dem Typ des Operanden, der kein Zeiger auf **void** ist.

Zuweisungs-Operatoren

Es gibt elf Zuweisungsoperatoren. Das Gleichheitszeichen (=) wird als einfacher, die anderen zehn als kombinierte Zuweisungsoperatoren bezeichnet.

Die Syntax eines Zuweisungsoperators lautet:

Zuweisungsausdruck:
Bedingter-Ausdruck
Unärer-Ausdruck Zuweisungsoperator Zuweisungsausdruck

Zuweisungsoperator: einer der folgenden
=*=/=%=+=-=
<<=>>=&=^=|=

Der einfache Zuweisungsoperator =

Im Ausdruck *E1* = *E2* muß der Operand *E1* ein modifizierbarer L-Wert sein. *E2* wird nach seiner Umwandlung in den Typ von *E1* in dem durch *E1* angegebenen Objekt gespeichert. Der frühere Wert von *E1* wird dadurch ersetzt. Der Wert des Zuweisungsausdrucks ist der Wert von *E1* nach der Zuweisung. Der Zuweisungsausdruck ist selbst kein L-Wert. Die Operanden *E1* und *E2* müssen einer der folgenden Regeln entsprechen:

1. *E1* ist ein qualifizierter oder unqualifizierter arithmetischer Typ und *E2* ist ein arithmetischer Typ.

2. *E1* ist eine qualifizierte oder unqualifizierte Version einer Struktur oder Variante, die mit dem Typ von *E2* kompatibel ist.

3. *E1* und *E2* sind Zeiger auf qualifizierte oder unqualifizierte Versionen kompatibler Typen; der Typ von *E1* hat alle Qualifizierer des Typs von *E2*.

4. Einer der Operanden *E1* und *E2* ist ein Zeiger auf ein Objekt oder einen unvollständigen Typ, der andere ein Zeiger auf eine qualifizierte oder unqualifizierte Version von **void**. Der Typ von *E1* hat alle Qualifizierer des Typs von *E2*.

5. *E1* ist ein Zeiger und *E2* ist die Null-Zeigerkonstante.

Hinweis: In C++ ist das Ergebnis ein L-Wert.

Die kombinierten Zuweisungsoperatoren

Die kombinierten Zuweisungsoperatoren haben die Form *op=*, wobei *op* eines der folgenden zehn Operator-Symbole sein muß: `* / % + - << >> & ^ |`.

Kombinierte Zuweisungsoperatoren haben folgende Syntax:

E1 op=E2

E1 =E1 op E2

hat denselben Effekt, allerdings wird dabei der L-Wert *E1* nur einmal ausgewertet. So ist etwa *E1 +=E2* identisch mit *E1 =E1 +E2*.

Für kombinierte Zuweisungsoperatoren gelten demnach dieselben Regeln wie für den einfachen Zuweisungsoperator = (siehe oben).

Der Komma-Operator

Seine Syntax lautet:

Ausdruck:
Zuweisungs-Ausdruck
Ausdruck , Zuweisungsausdruck

Im Komma-Ausdruck

E1 , E2

wird zunächst der linke Operand *E1* als **void**-Ausdruck ausgewertet. Die anschließende Auswertung von *E2* liefert das Ergebnis und den Typ des Komma-Ausdrucks. Der Ausdruck:

E1 , E2 , ... , En

wird für jedes *Ei* von links nach rechts ausgewertet, wobei der Wert und Typ von *En* das Ergebnis des gesamten Ausdrucks liefert. Zur Vermeidung von Mehrdeutigkeiten durch die in Funktionsargument- und Initialisierer-Listen enthaltenen Kommas müssen Klammern verwendet werden. Z.B. wird in

```
func(i, (j = 1, j + 4), k);
```

func mit drei Argumenten aufgerufen, nicht mit vier. Die Argumente sind *i*, *5* und *k*.

Hinweis: In C++ ist das Ergebnis ein L-Wert.

C++ Operatoren

Es gibt drei C++ spezifische Operatoren: `.*` `->*` und `::`. Der Gültigkeitsbereich-Zugriffsoperator `::` wird in Kapitel 3 »C++« beschrieben.

Die Syntax der Operatoren .* und ->* lautet wie folgt:

pm-Ausdruck
  cast-Ausdruck
  pm-Ausdruck .* *cast-Ausdruck*
  pm-Ausdruck ->* *cast-Ausdruck*

Der Operator .* dereferenziert Zeiger auf Klassenelemente. Er bindet den *cast-Ausdruck* an den *pm-Ausdruck*. Dabei muß der *cast-Ausdruck* vom Typ »Zeiger auf ein Element der Klasse *type*« sein, der *pm-Ausdruck* muß der Klasse *type* oder einer von *type* öffentlich abgeleiteten Klasse angehören. Das Ergebnis ist ein Objekt oder eine Funktion der durch den *cast-Ausdruck* spezifizierten Klasse.

Der Operator ->* dereferenziert Zeiger auf Zeiger auf Klassenelemente (dies ist kein Druckfehler – es werden tatsächlich Zeiger auf Zeiger dereferenziert). Er bindet den *cast-Ausdruck* an den *pm-Ausdruck*. Dabei muß der *cast-Ausdruck* vom Typ »Zeiger auf ein Element von *type*« sein, der *pm-Ausdruck* muß vom Typ »Zeiger auf *type*« oder »Zeiger auf eine von *type* öffentlich abgeleitete Klasse« sein. Das Ergebnis ist ein Objekt oder eine Funktion der durch den *cast-Ausdruck* spezifzierten Klasse.

Ist das Ergebnis eines dieser Operatoren eine Funktion, können Sie dieses Ergebnis nur als Operand für den Funktionsaufrufoperator ( ) verwenden. Das Beispiel

```
(ptr2object ->*ptr2memberfunc)  (10);
```

ruft die mit *ptr2memberfunc* angegebene Elementfunktion für das Objekt auf, auf das *ptr2object* zeigt.

Anweisungen

Anweisungen steuern den Ablauf während der Ausführung des Programms. Enthält das Programm keine spezifischen Sprung- und Auswahlanweisungen, werden die Anweisungen nacheinander, das heißt in der im Quelltext vorgegebenen Reihenfolge, ausgeführt.

Die folgende Tabelle zeigt eine Auflistung der Syntax der verschiedenen Anweisungen.

Tabelle 2.13 Turbo C++ Anweisungen

Anweisung:
Sprungziel
Blockanweisung
Ausdruckanweisung
Auswahlanweisung
Wiederholungsanweisung
Sprunganweisung
asm-Anweisung
Deklaration (C++)

asm-Anweisung:
asm *Symbole Zeilenvorschub*
asm *Symbole;*
asm { *Symbole;* <*Symbole;*>=
<*Symbole;*>
}

Sprungziel:
Bezeichner: Anweisung
case *Konstantenausdruck* : *Anweisung*
default : Anweisung

Blockanweisung:
{ <*Deklarationsliste*><*Anweisungsliste*>}

Deklarationsliste:
Deklaration
Deklarationsliste Deklaration

Anweisungsliste:
Anweisung
Anweisungsliste Anweisung

Ausdruckanweisung:
<*Ausdruck*>;

Auswahlanweisung:
if (*Ausdruck*) *Anweisung*
if (*Ausdruck*) *Anweisung* **else** *Anweisung*
switch (*Ausdruck*) *Anweisung*

Wiederholungsanweisung:
while (*Ausdruck*) *Anweisung*
do *Anweisung* **while** (*Ausdruck*) ;
for (*for-init-Anweisung* <*Ausdruck*>; <*Ausdruck*>) *Anweisung*

for-init-Anweisung:
Ausdruckanweisung
Deklaration (C++)

Sprunganweisung:
goto *Bezeichner* ;
continue ;
break ;
return <Ausdruck>;

Block-anweisungen

Eine Blockanweisung besteht aus einer (möglicherweise leeren) Liste von Anweisungen, die in geschweifte Klammern ({ }) gesetzt ist.

Syntaktisch kann eine Blockanweisung eine einzelne Anweisung sein, hat aber auch Auswirkungen auf den Gültigkeitsbereich von Bezeichnern. Der Gültigkeitsbereich eines in einer Blockanweisung deklarierten Bezeichners beginnt beim Deklarationspunkt und endet an der geschlossenen geschweiften Klammer. Blockanweisungen können beliebig tief geschachtelt werden.

Sprungziele

Eine Anweisung kann folgendermaßen mit einem Label (Sprungziel) versehen werden:

1. *Label-Bezeichner* : *Anweisung*

Der *Label-Bezeichner* dient als Ziel für die unbedingte **goto**-Anweisung. Derartige Bezeichner haben einen eigenen Namensbereich und ihr Gültigkeitsbereich erstreckt sich auf die ganze Funktion. In C++können Deklarations- und Nicht-Deklarationsanweisungen mit einem Label versehen werden.

2. **case** *Konstantenausdruck* : *Anweisung*
   default : *Anweisung*

Sprungziele in **case**- oder **default**-Anweisungen werden nur in Verbindung mit **switch**-Anweisungen eingesetzt.

Ausdruckanweisungen

Ein Ausdruck mit Strichpunkt bildet eine *Ausdruckanweisung*:

<Ausdruck>;

Die Ausführung einer Ausdruckanweisung erfolgt durch die Auswertung des Ausdrucks. Alle Nebeneffekte dieser Auswertung werden vor der Ausführung der nächsten Anweisung erzeugt. Bei den meisten Ausdruckanweisungen handelt es sich um Zuweisungsanweisungen oder Funktionsaufrufe.

Ein Sonderfall ist die *leere Anweisung* – sie besteht nur aus einem Strichpunkt (;). Die leere Anweisung hat zwar keine Auswirkungen, ist aber in manchen Fällen sinnvoll, zum Beispiel wenn die Syntax von Turbo C++eine Anweisung erwartet, das Programm aber keine benötigt.

Auswahlanweisungen

Auswahlanweisungen wählen aus alternativen Aktionsabläufen aus. Dies geschieht durch das Testen bestimmter Werte. Es gibt zwei Arten von Auswahlanweisungen: **if...else** und **switch**.

Die if-Anweisung
Die grundlegende Struktur von **if** sieht folgendermaßen aus:

if (*Bedingter-Ausdruck*) *w-Anweisung* <**else** *f-Anweisung*>

Die Klammern um *Bedingter-Ausdruck* sind unbedingt erforderlich. *Bedingter-Ausdruck* muß ein skalarer Typ sein. Der Ausdruck wird ausgewertet. Ist das Ergebnis 0, ist *Bedingter-Ausdruck* falsch; andernfalls ist er wahr.

Gibt es keine **else**-Klausel und ist *Bedingter-Ausdruck* wahr, so wird *w-Anweisung* ausgeführt; andernfalls wird *w-Anweisung* ignoriert.

Ist das optionale **else** *f-Anweisung* vorhanden und *Bedingter-Ausdruck* wahr, so wird *w-Anweisung* ausgeführt; andernfalls wird *w-Anweisung* ignoriert und *f-Anweisung* ausgeführt.

Hinweis: Im Gegensatz zu Sprachen wie Pascal hat Turbo C++ keinen spezifischen booleschen Datentyp. In Bedingungstests kann jeder Ausdruck vom Typ Integer oder Zeiger die Rolle eines booleschen Datentyps spielen. Der relationale Ausdruck $a > b$ wird (wenn er gültig ist) zu **int** 1 (wahr) ausgewertet, wenn $a > b$ ist. Ist dagegen $a <= b$, wird er zu **int** 0 (falsch) ausgewertet. Zeigerkonvertierungen werden so durchgeführt, daß ein Zeiger immer korrekt mit einem konstanten Ausdruck verglichen werden kann, der 0 ergibt. Der Test auf Null-Zeiger kann demnach auf zwei Arten geschrieben werden: `if (!ptr) ...` oder `if (ptr == 0)...`

Die Anweisungen *f-Anweisung* und *w-Anweisung* können selbst **if**-Anweisungen sein, wodurch verschachtelte Bedingungstests in beliebiger Tiefe ermöglicht werden. Bei verschachtelten **if...else**-Konstrukten muß sehr sorgfältig vorgegangen werden – es ist darauf zu achten, daß die korrekten Anweisungen ausgewählt werden. Eine **endif**-Anweisung gibt es nicht: **else**-Mehrdeutigkeiten werden gelöst, indem ein **else** dem letzten auf derselben Schachtelungstiefe aufgetretenen **if-ohne-else** zugeordnet wird. Beispielsweise zieht

```
if (x == 1)
   if (y == 1) puts("x=1 und y=1");
else puts("x != 1");
```

den falschen Schluß! Das **else** wird – trotz des Einzugs – dem zweiten **if** zugeordnet. Die korrekte Folgerung wäre: $x = 1$ und $y != 1$. Beachten Sie die Wirkung der geschweiften Klammern:

```
if (x == 1)
{
   if (y == 1) puts("x = 1 und y = 1");
}
else puts("x != 1");  // korrekte Folgerung
```

Die switch-Anweisung
Die **switch**-Anweisung hat das folgende Format:

switch (*sw-Ausdruck*) *case-Anweisungen*

Mit Hilfe einer **switch**-Anweisung kann die Steuerung an eine von mehreren Anweisungen übergeben werden, die mit einem **case**-Label versehen sind. Zu welchem **case**-Label gesprungen wird, ist vom Wert von *sw-Ausdruck* abhängig.

Der *sw-Ausdruck* muß ein Integertyp sein (in C++ ist auch ein Klassentyp zulässig, vorausgesetzt, es ist eine unzweideutige Konvertierung in einen Integertyp möglich).

Die *case-Anweisungen* (auch die leeren) können mit einem oder mehreren **case-**Labels versehen werden:

case *const-Ausdruck-i* : *case-Anweisung-i*

Jede **case**-Konstante *const-Ausdruck-i* ist ein konstanter Ausdruck mit einem eindeutigen Integerwert innerhalb der einschließenden **switch**-Anweisung. Er wird in den Typ des steuernden Ausdrucks umgewandelt.

Hinweis: Identische **case**-Konstanten innerhalb derselben **switch**-Anweisung sind unzulässig.

Es kann höchstens ein **default**-Label geben:

default : *default-Anweisung*

Nach der Auswertung von *sw-Ausdruck* wird eine Übereinstimmung mit einem der *const-Ausdruck-i* gesucht. Falls eine gefunden wird, geht die Steuerung an *case-Anweisung-i* mit dem passenden **case**-Label.

Wird keine Übereinstimmung gefunden und gibt es ein **default**-Label, erfolgt ein Sprung zu *default-Anweisung*. Schlagen sämtliche Vergleiche fehl und gibt es kein **default**-Label, wird keine der Anweisungen in *case-Anweisungen* ausgeführt.

Die weitere Programmausführung geht über folgende **case**- oder **default**-Labels einfach hinweg. Mit einer **break**-Anweisung kann man erreichen, daß der **switch**-Block am Ende einer Gruppe von Anweisungen zu einem **case**-Label verlassen wird.

Wiederholungs-anweisungen

Wiederholungsanweisungen ermöglichen die Erzeugung von Schleifen. In Turbo C++ gibt es drei Arten von Schleifen: **while**, **do** und **for**.

Die while-Anweisung

Das grundlegende Format für diese Anweisung lautet:

while (*Bedingungsausdruck*) *w-Anweisung*

Die Klammern um *Bedingungsausdruck* sind unbedingt erforderlich. Die Schleifenanweisung *w-Anweisung* wird solange ausgeführt, bis der *Bedingungsausdruck* den Wert 0 (falsch) ergibt. Der *Bedingungsausdruck* wird vorher ausgewertet und getestet (vgl. den Abschnitt »Auswahlanweisungen« weiter oben). Ergibt sich ein Wert, der ungleich 0 ist (wahr), wird *w-Anweisung* ausgeführt. Wenn das Programm auf keine Sprunganweisungen trifft, die das Verlassen der Schleife zur Folge haben, wird der *Bedingungsausdruck* erneut ausgewertet. Dieser Vorgang wiederholt sich solange, bis *Bedingungsausdruck* den Wert 0 ergibt.

Ähnlich wie bei der **if**-Anweisung können Ausdrücke vom Typ Zeiger mit dem Null-Zeiger verglichen werden; demnach sind `while (ptr) ...` und `while (ptr != NULL)...` identisch.

while-Schleifen stellen eine gute Methode zum Durchsuchen von Strings und anderen null-terminierten Datenstrukturen dar:

```
char str[10]="Borland";
char *ptr=&str[0];
int count=0;
//...
while (*ptr++)   // Schleife endet mit dem Schluß des Strings
   count++;
```

Gibt es keine Sprunganweisungen, muß *w-Anweisung* den Wert von *Bedingungsausdruck* beeinflussen oder *Bedingungsausdruck* selbst muß sich während der Auswertung ändern, damit Endlosschleifen vermieden werden.

Die do while-Anweisung

Das Format für eine **do while**-Anweisung lautet:

do *do-Anweisung* **while** (*Bedingungsausdruck*);

Die Anweisung *do-Anweisung* wird solange ausgeführt, bis *Bedingungsausdruck* den Wert 0 (falsch) hat.

Im Gegensatz zur **while**-Anweisung wird *Bedingungsausdruck* nicht vor, sondern *nach* jeder Ausführung der Schleifenanweisung getestet. *do-Anweisung* wird daher mindestens einmal ausgeführt.

Für *Bedingungsausdruck* gelten dieselben Regeln wie bei der **while**-Anweisung.

Die for-Anweisung

Das Format für eine **for**-Anweisung lautet:

for (*<Initialisierungsausdruck>* ; *<Testausdruck>* ;
<Inkrementierungsausdruck>) *Anweisung;*

Hinweis: In C++ kann *Initialisierungsausdruck* ein Ausdruck oder eine Deklaration sein.

for-Anweisungen werden folgendermaßen abgearbeitet:

1. Der *Initialisierungsausdruck* wird, falls vorhanden, ausgeführt. Dadurch werden ein oder mehrere Schleifenzähler initialisiert. Die Syntax ermöglicht Ausdrücke von höchster Komplexität (einschließlich Deklarationen in C++). Daher der Anspruch, daß jedes C-Programm als eine einzige **for**-Schleife geschrieben werden kann. (Versuchen Sie dies aber bitte nicht.)

2. Der Ausdruck *Testausdruck* wird ausgewertet (es gelten dieselben Regeln wie für **while**-Schleifen). Ergibt *Testausdruck* einen Wert, der ungleich 0 ist (wahr), wird die Schleifenanweisung ausgeführt. Ein leerer Ausdruck wird hier wie **while**(1), also als wahr interpretiert. Ist der Wert von *Testausdruck* 0 (falsch), endet die **for**-Schleife.

3. *Inkrementierungsausdruck* erhöht einen oder mehrere Zähler.

4. Die *Anweisung* (sie kann leer sein) wird ausgeführt, und die Steuerung kehrt zu Schritt 2 zurück.

Sind ein oder mehrere optionale Elemente leer, müssen entsprechende Strichpunkte gesetzt werden:

```
for (;;) {                  //entspricht for (; 1;)
// Endlosschleife
}
```

In C++ gelten die C-Regeln für **for**-Anweisungen. Der *Initialisierungsausdruck* kann in C++ aber auch eine Deklaration sein.

Der Gültigkeitsbereich eines deklarierten Bezeichners endet mit der Schleifenanweisung:

```
for (int i = 1; i < j; ++i)
{
   if (i ...) ...           // richtig: Bezug auf i ist hier
                            // erlaubt
}
if (i...)                   // falsch: i liegt jetzt außerhalb
                            // des Gültigkeitsbereichs
```

Sprunganweisungen

Bei der Ausführung einer Sprunganweisung wird die Programmsteuerung unbedingt übertragen. Folgende Sprunganweisungen stehen in Turbo C++ zur Verfügung: **break**, **continue**, **goto** und **return**.

Die break-Anweisung

Die Syntax für die **break**-Anweisung lautet folgendermaßen:

break ;

Die **break**-Anweisung kann nur innerhalb von Schleifen (**while**, **do** oder **for**) oder einer **switch**-Anweisung verwendet werden und dient zur Beendigung der Schleife bzw. der Anweisung. Da Schleifen und **switch**-Anweisungen gemischt und beliebig tief geschachtelt sein können, ist darauf zu achten, daß die **break**-Anweisung für die richtige Schleife oder Anweisung eingesetzt wird. Es gilt folgende Grundregel: **break** terminiert die innerste einschließende Schleife oder **switch**-Anweisung.

Die continue-Anweisung

Die Syntax für diese Anweisung lautet folgendermaßen:

continue;

Die **continue**-Anweisung kann nur innerhalb von Schleifen verwendet werden. In **while**- und **do**-Schleifen überträgt sie die Steuerung an die Testbedingung, in einer **for**-Schleife an den Inkrement-Ausdruck.

In verschachtelten Schleifen wird eine **continue**-Anweisung als zur innersten einschließenden Schleife gehörig betrachtet.

Die Anweisung goto
Die Syntax für diese Anweisung lautet folgendermaßen:

goto *label;*

Die **goto**-Anweisung überträgt die Programmsteuerung zur Anweisung mit dem Sprungziel *label* (vergleichen Sie den Abschnitt »Sprungziele« weiter oben). Das Sprungziel muß sich innerhalb derselben Funktion befinden.

In C++ darf durch *goto* eine Deklaration mit einem expliziten oder impliziten Initialisierer nicht übergangen werden. Dies gilt allerdings nicht, wenn sich die Deklaration in einem inneren Block befindet, der ebenfalls übergangen wird.

Die Anweisung return
Ein Funktionsrumpf muß mindestens eine **return**-Anweisung enthalten (ausser der zurückgelieferte Typ der Funktion ist **void**). Die Syntax für die **return**-Anweisung lautet folgendermaßen:

return *return-Ausdruck* ;

return-Ausdruck muß vom Typ *type* sein (oder von einem Typ, der zu *type* konvertiert werden kann). Der Wert von *return-Ausdruck* ist der Wert, der von der Funktion zurückgeliefert wird. Ein Ausdruck zum Aufruf einer Funktion, zum Beispiel `func(actual-arg-list)` , ist kein L-Wert, sondern ein R-Wert vom Typ *type*:

```
t = func(arg);     // OK
func(arg) = t;     // illegal in C; legal in C++, wenn der
                   // zurückgelieferte Typ von func eine
                   // Referenz ist
(func(arg))++;     // illegal in C; legal in C++, wenn der
                   // zurückgelieferte Typ von func eine
                   // Referenz ist
```

Trifft das Programm auf eine **return**-Anweisung, wird die Ausführung eines Funktionsaufrufs beendet. Ist dagegen kein **return** vorhanden, endet die Ausführung bei der letzten geschlossenen geschweiften Klammer im Funktionsrumpf.

Ist der zurückgelieferte Typ **void**, kann die **return**-Anweisung folgendermaßen auch ohne *return-Ausdruck* geschrieben werden:

```
{
    ...
    return;
}
```

Alternativ dazu kann die **return**-Anweisung auch weggelassen werden.

3. C++

C++ ist grundsätzlich eine Obermenge von C. Es ist daher möglich, C-Programme in einer C++ Umgebung zu compilieren. Die Übernahme von C++ Programmen in eine C-Umgebung funktioniert aber nur, wenn keine speziellen Eigenschaften von C++ verwendet werden.

Einige Situationen erfordern besondere Vorsicht. Wird z.B. eine Funktion gleichen Namens in C zweimal mit verschiedenen Argumenttypen definiert, erzeugt dies einen Fehler. In C++ wird die Funktion dann als überladene Funktion interpretiert – ob dies erlaubt ist oder nicht, hängt von bestimmten Voraussetzungen ab.

Obwohl C++ neue Schlüsselwörter und Operatoren zur Handhabung von Klassen eingeführt hat, können einige Möglichkeiten von C++ auch außerhalb von Klassen verwendet werden. Wir wollen uns zunächst mit den Aspekten von C++ befassen, die unabhängig von Klassen benutzt werden können.

Referenzen

Während in C Argumente nur durch einen Wert übergeben werden können, ist es in C++ möglich, Argumente als Wert oder als Referenz auf diesen Wert zu übergeben. Die C++ Referenztypen sind den Zeigertypen sehr ähnlich. Sie erzeugen Stellvertreter für Objekte, die zur Übergabe von Argumenten an Funktionen benutzt werden können. (Näheres zum Thema Referenzieren und Dereferenzieren finden Sie in Kapitel 2 »Semantische Grammatik« im Abschnitt »Unäre Operatoren«).

Einfache Referenzen

Der Referenzdeklarator kann dazu benutzt werden, Referenzen außerhalb von Funktionen zu definieren:

```
int i = 0;
int &ir = i;   // ir ist ein Stellvertreter für i
ir = 2;        // hat den gleichen Effekt wie i = 2
```

Im Beispiel wird der L-Wert *ir* als Stellvertreter für *i* deklariert, wobei der Initialisierer vom gleichen Typ wie die Referenz sein muß. Jede Operation mit *ir* hat denselben Effekt wie die Operation mit *i*. So weist `ir = 2` beispielsweise *i* den Wert 2 zu, und `&ir` liefert die Adresse von *i* zurück.

Hinweis: type& var, *type &var* und *type & var* sind identische Deklarationen.

Referenzen als Argumente

Der Referenzdeklarator kann auch dazu benutzt werden, um Referenzen als Übergabeparameter einer Funktion zu definieren:

```
void func1 (int i);
void func2 (int &ir);  // ir ist vom Typ "Referenz auf einen int"
   ...
int sum=3;
func1(sum);            // sum wird als Wert übergeben
func2(sum);            // sum wird als Referenz übergeben
```

Das als Referenz übergebene Argument *sum* kann von der Funktion *func2* direkt geändert werden. Dagegen erhält *func1* nur eine Kopie des Wertes von *sum* und kann so *sum* selbst nicht ändern.

Wird ein Argument *x* durch seinen Wert übergeben, erhält der formale Parameter innerhalb der Funktion eine Kopie des Wertes von *x*. Eine Änderung dieser Kopie innerhalb der aufgerufenen Funktion hat keine Auswirkung auf den Wert von *x* selbst. Natürlich kann die Funktion einen Wert *zurückgeben*, der später zur Veränderung von *x* benutzt werden kann. Die Funktion kann die Änderung allerdings nicht selbst durchführen.

Die traditionelle C-Methode, der aufgerufenen Funktion die Änderung von *x* zu erlauben, ist die Übergabe des Zeigers *&x*. Obwohl der Zeiger als Wert übergeben wird, kann die Funktion doch über die Kopie des Zeigers *x* selbst beeinflussen. Selbst wenn *x* gar nicht von der Funktion geändert werden muß, ist es oft nützlich, einen Zeiger auf *x* zu übergeben (es kann dadurch allerdings zu unerwünschten Nebeneffekten kommen). Dies gilt speziell dann, wenn *x* eine große Datenstruktur ist. Die Übergabe des Zeigers spart dann die aufwendige Kopie der Daten.

Vergleichen Sie die drei folgenden Implementationen der Funktion *treble*:

Implementation 1:

```
int treble_1(n)
{
   return 3*n;
}
...
int x, i = 4;
x = treble_1(i);       // x = 12, i = 4
...
```

Implementation 2:

```
void treble_2(int* np)
{
   *np = (*np)*3;
}
...
treble_2(int &i);      // i = 12
```

Implementation 3:

```
void treble_3(int& n)  // n ist ein Referenztyp
{
   n = 3*n;
}
...
treble_3(i);           // i = 36
```

Die formale Parameterdeklaration *type& t* (entspricht *type &t*) definiert *t* als Typ »Referenz auf *type*«. Wird demnach *treble_3* mit dem Argument *i* aufgerufen, dient *i* zur Initialisierung des Referenzparameters *n*. *n* agiert als Stellvertreter von *i*, so daß die Zuweisung *n = 3*n* gleichbedeutend mit der Zuweisung von *3 * i* auf *i* ist. Ist der Initialisierer eine Konstante oder ein Objekt mit einem anderen Typ als der Referenztyp, erzeugt Turbo C++ ein temporäres Objekt, für das die Referenz als Stellvertreter agiert:

```
int& ir = 6;           /* erzeugt ein temporäres Integerobjekt,
                          für das ir als Stellvertreter benutzt wird
                          und das den Wert 6 hat */
float f;
int& ir2 = f;          /* erzeugt ein temporäres Integerobjekt,
                          für das ir2 der Stellvertreter ist;
                          f wird vor der Zuweisung in einen
                          Integer umgewandelt */
ir2 = 2.0              /* ir2 bekommt nun den Wert 2,
                          f bleibt unverändert */
```

Die automatische Erzeugung von temporären Objekten ermöglicht die Konvertierung von Referenztypen, wenn die formalen und tatsächlichen Argumente verschiedene (aber zuweisungskompatible) Typen haben. Bei der Wertübergabe gibt es allerdings weniger Konvertierungsprobleme, weil die Kopie des aktuellen Arguments physikalisch geändert werden kann, bevor sie dem formalen Parameter zugewiesen wird.

Der Gültigkeitsbereich-Zugriffsoperator

Der Gültigkeitsbereich-Zugriffsoperator (kurz: Zugriffsoperator oder Auflösungsoperator) `::` (zwei Doppelpunkte) erlaubt Ihnen, auf einen globalen

Namen zuzugreifen, selbst wenn er durch eine lokale Deklaration (desselben Namens) verborgen ist.

Hinweis: Ausführliche Informationen zum Gültigkeitsbereich finden Sie in Kapitel 1 »Lexikalische Grammatik«.)

```
int i;                    // globales i
...

void func(void);

   {
      int i=0;            // lokales i verbirgt globales i
      i = 3;              // dieses i ist das lokale i
      ::i = 4;            // dieses i ist das globale i
      printf ("%  d",i);    // gibt 3 aus
}
```

Dieses Programm funktioniert auch, wenn die »globale« Variable eine statische Variable auf Dateiebene ist.

Der Operator :: wird mit Klassen in einem anderen Zusammenhang benutzt. Im folgenden wird darauf näher eingegangen.

Die Operatoren new und delete

Die Operatoren **new** und **delete** ermöglichen die dynamische Belegung und Freigabe von Speicher, ähnlich (aber wirkungsvoller) wie die Standardbibliotheksfunktionen der Familien *malloc* und *free* (siehe *Referenzhandbuch*). Vereinfacht lautet ihre Syntax folgendermaßen:

Zeiger-auf-Name new Name <Name-Initialisierer>;
delete Zeiger-auf-Name ;

wobei *Name* jeden Typ außer dem Funktionstyp haben kann (allerdings sind Zeiger auf Funktionen erlaubt).

new versucht, ein Objekt des Typs *Name* anzulegen, indem es einen der Größe entsprechenden Speicherbereich im freien Speicher (dem Heap) belegt. Die Belegung dieses Speicherbereichs besteht, bis **delete** den Speicher wieder freigibt oder das Programm beendet wird.

Ist die Speicherplatzbelegung erfolgreich, gibt **new** einen Zeiger auf das neue Objekt zurück. Andernfalls (zum Beispiel bei ungenügendem oder fragmentiertem Heap-Speicher) liefert **new** einen Null-Zeiger zurück. Wie bei *malloc* muß vor der Benutzung des neuen Objekts dieser Zeiger auf Null getestet werden (außer Sie verwenden einen *new*-Handler).

Im Gegensatz zu *malloc* errechnet **new** den nötigen Speicherplatzbedarf für das Objekt selbst. Außerdem ist der zurückgegebene Zeiger vom korrekten Typ »Zeiger auf *Name*«, ohne daß eine Typumwandlung (cast) erforderlich wäre.

Da **new** ein Schlüsselwort ist, wird kein Prototyp dafür benötigt.

```
name *nameptr;      // name kann jeder Typ außer einer Funktion sein
...
if (!(nameptr = new name))  {
   errmsg("Nicht genug Speicher für name");
   exit (1);
}
// verwenden Sie *nameptr, um das neue Objekt zu initialisieren
...
delete nameptr;     // löscht name und gibt sizeof(name) Bytes frei
```

Fehler-behandlung

Sie können eine Funktion definieren, die aufgerufen wird, wenn der **new**-Operator erfolglos bleibt (0 zurückgibt). Damit **new** diesen *new*-Handler verwenden kann, rufen Sie *set_new_handler* auf und übergeben einen Zeiger auf den *new*-Handler.

Der Prototyp für *set_new_handler* sieht folgendermaßen aus (aus new.h):

```
void (*set_new_handler( void (*) () )) ();
```

set_new_handler liefert den alten *new*-Handler zurück und ändert die Funktion *_new_handler* so, daß sie auf den von Ihnen definierten *new*-Handler zeigt.

Der Operator new im Zusammenhang mit Arrays

Ist das Objekt *name*, auf das **new** angewendet wird, ein Array, liefert **new** einen Zeiger zurück, der auf das erste Element des Array zeigt.

Soll mit **new** ein mehrdimensionales Array angelegt werden, müssen alle Dimensionen des Array angegeben werden (obwohl die am weitesten links stehende Dimension keine Compilierzeit-Konstante sein muß):

```
mat_ptr = new int[3][10][12];      // richtig
mat_ptr = new int[n][10][12];      // richtig
mat_ptr = new int[3][][12];        // falsch
mat_ptr = new int[][10][12];       // falsch
```

Obwohl die erste Array-Dimension eine Variable sein kann, müssen alle nachstehenden Dimensionen Konstanten sein.

Der Operator delete im Zusammenhang mit Arrays

Wenn Sie einen Array löschen, müssen Sie die Syntax **delete**[] expr verwenden. In der C++ Version 2.1 sollte die Array-Dimension nicht in den eckigen Klammern stehen:

```
char * p;

void func()
{
   p = new char[10];    // reserviert 10 Zeichen
   delete[] p;          // löscht 10 Zeichen
}
```

Programmtext in der C++ Version 2.0 erfordert die Array-Größe. Damit Programmtext der Version 2.0 compiliert werden kann, gibt Turbo C++ eine Warnung aus und ignoriert die angegebene Größe.

Wenn beispielsweise im vorangehenden Programmtext **delete**[10] stehen und dieser compiliert würde, würde die Warnung folgendermaßen lauten:

```
Warning: Array size for 'delete'ignored in function func()
```

In Turbo C++ist [] nur dann erforderlich, wenn das Array-Element eine Klasse mit einem Destruktor ist. Es gehört aber zu gutem Programmierstil, dem Compiler mitzuteilen, daß ein Array gelöscht wird.

Der ::operator new

Wenn **new** in Zusammenhang mit Objekten verwendet wird, die keine Klassen sind, benutzt **new** eine Standardbibliotheksroutine, den globalen ::*operator new*. Für Objekte einer Klasse vom Typ *Name* kann ein spezieller Operator, der *Name*::*operator new* genannt wird, definiert werden.

Bei einer Anwendung von **new** auf diese Objekte wird der vorher definierte *Name::operator new* verwendet; wurde dieser nicht definiert, wird der normale ::*operator new* verwendet.

Initialisierer in Zusammenhang mit dem Operator new

Die Möglichkeit, Initialisierer anzugeben, ist ein weiterer Vorteil, den **new** gegenüber der Verwendung von *malloc* hat (obwohl *calloc* den reservierten Speicher mit Null initialisiert).

Wird kein Initialisierer angegeben, hat das von **new** erzeugte Objekt einen unvorhersehbaren (höchstwahrscheinlich unbrauchbaren) Inhalt.

Deswegen können alle Objekte, die von **new** erzeugt werden und keine Arrays sind, mit einem in Klammern gesetzten Ausdruck vorbelegt werden:

```
int_ptr = new int(3);
```

Arrays von Klassen, die einen Konstruktor haben, werden mit dem *Standardkonstruktor* initialisiert (vgl. den Abschnitt »Der Standardkonstruktor« weiter hinten in diesem Kapitel). Der benutzerdefinierte Operator **new** mit einer vom Benutzer vorgegebenen Initialisierung spielt eine Schlüsselrolle bei C++ Konstruktoren für Klassen.

Klassen

C++ Klassen bieten eine Erweiterung des vordefinierten Typsystems. Jeder Klassentyp repräsentiert eine eindeutige Menge von Objekten und stellt Operationen (Methoden) und Konvertierungen zur Verfügung, um diese Objekte zu erzeugen, zu bearbeiten und zu löschen. Es können abgeleitete Klassen definiert werden, die die Elemente einer oder mehrerer *Basisklassen* erben.

In C++ werden Strukturen und Varianten als Klassen mit speziellen standardmäßigen Zugriffsmöglichkeiten betrachtet.

Eine vereinfachte Syntax für Klassendeklarationen lautet:

Klassenschlüssel Klassenname <: *Basisliste*>{ <*Elementliste*>}

Klassenschlüssel kann eines der reservierten Wörter **class**, **struct** oder **union** sein. Die – zusammen mit dem Doppelpunkt optionale – *Basisliste* zählt die Basisklasse(n) auf, von denen *Klassenname* Objekte und Methoden erben soll. Wenn zumindest eine Basisklasse spezifiziert ist, wird die neue Klasse *Klassenname* »abgeleitete Klasse« genannt (vgl. »Der Zugriff auf Basisklassen und abgeleitete Klassen« weiter hinten in diesem Kapitel).

Die verwendeten Basisklassen haben voreingestellte *Zugriff-Spezifizierer*, die auch explizit angegeben werden können, um den Zugriff der abgeleiteten Klasse auf Elemente der Basisklasse zu beeinflussen (vgl. »Element-Zugriffskontrolle«).

Die optionale *Elementliste* deklariert die Klassenelemente (Daten und Funktionen) von *Klassenname* mit standardmäßigen und optionalen, expliziten Zugriff-Spezifizierern, und legt so fest, welche Funktionen auf welche Elemente zugreifen dürfen.

Klassennamen

Als *Klassenname* kann jeder Bezeichner verwendet werden, der innerhalb seines Gültigkeitsbereichs eindeutig ist. Bei Strukturen, Klassen und Varianten kann der Klassenname weggelassen werden (vgl. »Strukturen und Typdefinitionen ohne »Tag«-Name« in Kapitel 2 »Semantische Grammatik«).

Klassentypen

Die folgende Deklaration erzeugt den eindeutigen Klassentyp *Klassenname*. Damit können Sie weitere Objekte dieses Typs (sog. *Instanzen*) und von diesem Typ abgeleitete Objekte (z.B. Zeiger auf *Klassenname*, Referenzen auf *Klassenname* oder Arrays von *Klassenname*) definieren:

```
class X { ... };
X x, &xr, *xptr, xarray[10];
// vier Objekte:
// Typ X, Referenz auf X, Zeiger auf X und Array von X
```

```
struct Y { ... };
Y y, &yr, *yptr, yarray[10];
// in C wäre das:
// struct Y y, &yr, *yptr, yarray[10];

union Z { ... };
Z z, &zr, *zptr, zarray[10];
// in C wäre das:
// union Z z, &zr, *zptr, zarray[10];
```

Beachten Sie den Unterschied zwischen C- und C++Struktur- und Variantendeklarationen: Die Schlüsselwörter **struct** und **union** sind in C notwendig, in C++ werden sie nur gebraucht, wenn die Klassennamen – im obigen Beispiel *Y* und *Z* – nicht sichtbar sind.

Der Gültigkeitsbereich von Klassennamen

Klassennamen haben, abgesehen von einigen Besonderheiten, einen lokalen Gültigkeitsbereich. Er beginnt bei der Deklaration und endet mit dem einschließenden Block. Ein Klassenname wird vom selben Namen einer anderen Klasse, eines Objekts, eines Aufzählungstyps oder einer Funktion mit demselben Namen im Gültigkeitsbereich des einschließenden Blocks verdeckt. Wird ein Klassenname in einem Gültigkeitsbereich definiert, der die Definition eines Objekts, einer Funktion oder eines Aufzählungstyps mit demselben Namen enthält, kann auf diese Klasse nur mit Hilfe seines *Typ-Spezifizierers* zugegriffen werden. Das bedeutet, der Klassenschlüssel **class**, **struct** oder **union** muß zusammen mit dem Klassennamen verwendet werden. Ein Beispiel:

```
struct S { ... };
int S(struct S *Sptr);
void func(void)
{
   S t;        // ILLEGALE Deklaration: kein Klassenschlüssel
               // und Funktion S im Gültigkeitsbereich
   struct S s; // OK: mit Klassenschlüssel
   S(&s);      // OK: das ist ein Funktionsaufruf
```

C++ erlaubt außerdem die unvollständige Deklaration einer Klasse:

```
class X;       // noch keine Elemente
struct Y;
union Z;
```

Unvollständige Deklarationen von Klassen ermöglichen die Nutzung spezieller Referenzen auf den Klassennamen *X* (normalerweise Referenzen auf Zeiger auf die Klassenobjekte), bevor die Klasse vollständig deklariert wird (vgl. Kapitel 2 »Semantische Grammatik«). Natürlich muß die Klasse vollständig deklariert sein, bevor Klassenobjekte definiert und genutzt werden können.

Klassen-objekte

Klassenobjekte können auf der linken Seite von Zuweisungen verwendet (falls das Kopieren erlaubt ist), als Argumente an Funktionen übergeben und – mit Einschränkungen – von Funktionen zurückgegeben werden, etc.

Viele andere Operationen mit Klassenobjekten und -elementen können vom Benutzer definiert werden, z.B. durch Element- und Friend-Funktionen oder das Überschreiben von Standardfunktionen und -operatoren, wenn sie mit Objekten einer bestimmten Klasse verwendet werden.

Operatoren und Funktionen, die in ihrer Gültigkeit auf eine spezielle Klasse (oder auf verknüpfte Klassengruppen) beschränkt sind, werden *Elementfunktionen* dieser Klasse genannt. C++ verfügt über einen Mechanismus, mit dem derselbe Funktions- oder Operatorname aufgerufen werden kann, um verschiedene – vom Typ oder von der Anzahl der Argumente oder Operanden abhängige – Aufgaben auszuführen, das sogenannte Überladen von Funktionen.

Die Klassen-elementliste

Die optionale *Elementliste* ist eine Folge von Datendeklarationen (jeden Typs, einschließlich Aufzählungstypen, Bitfelder und anderer Klassen) und Funktionsdeklarationen und -definitionen, für die Speicherklassen-Spezifizierer und Zugriff-Modifizierer verwendet werden können. Die auf diese Weise definierten Objekte werden *Klassenelemente* genannt. Die Speicherklassen-Spezifizierer **auto**, **extern** und **register** sind nicht erlaubt, **static** und **volatile** sind zulässig (**static** hat eine besondere Bedeutung).

Element-funktionen

Eine Funktion, die ohne den **friend**-Spezifizierer deklariert wird, wird als *Elementfunktion* der Klasse bezeichnet. Funktionen mit dem **friend**-Spezifizierer heißen *Friend-Funktionen*.

Mehreren Funktionen kann derselbe Name gegeben werden. Voraussetzung dafür ist aber, daß sie sich im Typ oder in der Anzahl der Argumente unterscheiden.

Das Schlüssel-wort this

Nicht-statische Elementfunktionen operieren mit dem Klassenobjekt, mit dem sie aufgerufen werden. Ist z.B. *x* ein Objekt der Klasse *X* und *f* eine Elementfunktion von *X*, operiert der Funktionsaufruf `x.f()` mit *x*. Ist *xptr* ein Zeiger auf ein Objekt von *X*, operiert der Funktionsaufruf `xptr->f()` mit `*`*xptr*.

Wie weiß nun aber die Funktion *f*, mit welchem *x* sie operiert? C++ übergibt *f* einen Zeiger auf *x*, und zwar mit dem Namen **this**. **this** wird als verborgenes Argument an alle nicht-statischen Elementfunktionen übergeben.

Das Schlüsselwort **this** steht für eine lokale Variable, die in jeder nicht-statischen Elementfunktion verwendbar ist. **this** muß nicht deklariert werden und wird in einer Funktionsdefinition nur selten explizit verwendet.

Innerhalb der Funktion wird **this** jedoch implizit verwendet, um Elemente zu referenzieren. Wird zum Beispiel $x.f(y)$ aufgerufen, wobei *y* ein Element von *X* ist, erhält **this** den Wert *&x*, und *y* wird zu *this->y*, was gleichbedeutend mit *x.y* ist.

Inline-Funktionen

Sie können eine Elementfunktion innerhalb einer Klasse deklarieren und an anderer Stelle definieren. Alternativ dazu ist es möglich, die Elementfunktion innerhalb einer Klasse zu deklarieren *und* zu definieren. In diesem Fall wird die Funktion *Inline*-Funktion genannt.

Turbo C++ kann manchmal den normalen Funktionsaufruf-Overhead vermeiden, indem es anstelle des Funktionsaufrufs direkt den compilierten Code der Funktion einsetzt. Dieser Vorgang, der *Inline-Ausdehnung* der Funktion genannt wird, hat keinen Einfluß auf den Gültigkeitsbereich des Funktionsnamens oder seiner Argumente.

Eine Inline-Ausdehnung ist aber nicht immer möglich oder nützlich. Der **inline**-Spezifizierer ist ein Hinweis für den Compiler, daß Sie eine Inline-Ausdehnung wünschen. Wie beim Speicherklassen-Spezifizierer **register** ist der Compiler allerdings nicht gezwungen, diesem Hinweis zu folgen.

Explizite und implizite Inline-Anweisungen werden am besten nur für kleine, häufig benutzte Funktionen verwendet, wie z.B. Operatorfunktionen, die überladene Operatoren implementieren.

Ein Beispiel: die Klassendeklaration

```
int i;                                  // globales int

class X {
public:
   char* func(void) { return i; }   // standardmäßig inline
   char* i;
};
```

ist gleichbedeutend mit

```
inline char* X::func(void) { return i; }
```

func ist »außerhalb« der Klasse mit dem expliziten **inline**-Spezifizierer definiert. Das von *func* zurückgegebene *i* ist der **char*i** der Klasse *X* (vgl. den Abschnitt »Der Gültigkeitsbereich von Elementen« weiter hinten in diesem Kapitel).

Statische Elemente

Der Speicherklassen-Spezifizierer **static** kann in Deklarationen von Daten- und Funktionselementen verwendet werden. Derartige Elemente heißen *statische Elemente* und unterscheiden sich in bestimmten Eigenschaften von den nicht-statischen Elementen. Bei nicht-statischen Elementen existiert für jedes Objekt seiner Klasse eine eigene Kopie; bei statischen Elementen gibt es nur eine Kopie, auf die ohne Referenz auf ein bestimmtes Objekt seiner Klasse zugegriffen werden kann. Ist *x* ein statisches Element der Klasse *X*, kann es als *X::x* referenziert werden (auch dann, wenn noch keine Objekte der Klasse *X* erzeugt wurden). Es ist jedoch auch möglich, unter Benutzung der normalen Operatoren auf *x* zuzugreifen, z.B. *y.x* und *yptr->x*, wobei *y* ein Objekt der Klasse *X* und *yptr* ein Zeiger auf ein Objekt der Klasse *X* ist, obwohl die Ausdrücke *y* und *yptr* nicht ausgewertet werden. Eine statische Elementfunktion kann mit oder ohne spezielle Elementfunktions-Syntax aufgerufen werden:

```
class X {
   int member_int;
public:
   static void func(int i, X* ptr);
};
void g(void);
{
   X obj;
   func(1, &obj);          // Fehler, wenn es keine globale
                           // func() gibt
   X::func(1, &obj);       // Ruft die statische func() in X auf
                           // (nur für statische Funktionen)
   obj.func(1, &obj);      // Ruft die statische func() in X auf
                           // (für statische und nicht-statische
                           // Funktionen)
}
```

Da eine statische Elementfunktion ohne ein spezielles Objekt aufgerufen werden kann, hat sie keinen **this**-Zeiger. Eine statische Elementfunktion kann deshalb *nicht* ohne Spezifizierung eines Objekts mit . oder -> auf nicht-statische Elemente zugreifen. Wenn wir das vorherige Beispiel zugrunde legen, könnte *func* z.B. wie folgt definiert werden:

```
void X::func(int i, X* ptr)
{
   member_int = i;         // Auf welches Objekt bezieht
                           // sich member_int? Fehler!
   ptr->member_int = i;    // OK: Jetzt wissen wir Bescheid!
}
```

Abgesehen von Inline-Funktionen haben statische Elementfunktionen von globalen Klassen externe Bindung. Statische Elementfunktionen können keine virtuellen Funktionen sein. Es ist nicht erlaubt, eine statische und eine nicht-

statische Elementfunktion mit demselben Namen und denselben Argumenttypen zu versehen.

Die Deklaration eines statischen Datenelements innerhalb seiner Klasse ist noch keine Definition. Die Definition muß deshalb an anderer Stelle vorgenommen werden, um Speicherplatz zu reservieren und eine Initialisierung durchzuführen.

Statische Elemente einer Klasse, die lokal zu einer Funktion deklariert ist, haben keine Bindung und können nicht initialisiert werden. Statische Elemente einer globalen Klasse können wie normale globale Objekte initialisiert werden, allerdings nur im Gültigkeitsbereich ihrer Datei.

Abgesehen davon, daß sie initialisiert werden können, gehorchen statische Elemente den normalen Zugriffsregeln für Klassenelemente.

```
class X {
   ...
   static int x;
   ...
};
int X::x = 1;
```

Der Hauptzweck statischer Elemente ist es, Daten aufzunehmen, die allen Objekten einer Klasse gemeinsam sind (z.B. die Anzahl von erzeugten Objekten oder die zuletzt verwendete Ressource aus einem Pool, aus dem sich alle diese Objekte bedienen). Statische Elemente werden außerdem genutzt, um

- die Anzahl der sichtbaren globalen Namen zu reduzieren,
- offensichtlich zu machen, welches statische Objekt logisch zu welcher Klasse gehört,
- den Zugriff auf ihre Namen zu steuern.

Der Gültigkeitsbereich von Elementen

Der Ausdruck *X::func()* im ersten Beispiel zu den Inline-Funktionen verwendet den Klassennamen *X* mit dem Zugriffsoperator, um zu zeigen, daß *func* – obwohl außerhalb seiner Klasse definiert – trotzdem eine Elementfunktion von *X* ist und innerhalb des Gültigkeitsbereichs von *X* existiert. Der Einfluß von *X::* wirkt auch innerhalb der Funktionsdefinition. Das erklärt, warum sich das von *func* zurückgelieferte *i* auf *X::i* (dem *char* i* von *X*) bezieht, und nicht auf den globalen *int i*. Ohne den Modifizierer *X::* wäre die Funktion *func* eine normale Funktion, die keiner Klasse angehört, und würde folglich den globalen *int i* zurückliefern.

Alle Elementfunktionen sind also im Gültigkeitsbereich ihrer Klasse, auch dann, wenn sie außerhalb der Klasse definiert wurden.

Auf Datenelemente der Klasse *X* kann mit Hilfe der Auswahloperatoren . und -> zugegriffen werden (wie in C-Strukturen). Dies gilt auch für Elementfunktionen (vgl. den Abschnitt »Das Schlüsselwort **this**« weiter vorne). Ein Beispiel:

```
class X {

public:
   int i;
   char name[20];
   X *ptr1;
   X *ptr2;
   void Xfunc(char*data, X* left, X* right); // anderswo definiert
};
void f(void);
{
   X x1, x2, *xptr=&x1;
   x1.i = 0;
   x2.i = x1.i;
   xptr->i = 1;
   x1.Xfunc("stan", &x2, xptr);
}
```

Für ein Element oder Basiselement *m* der Klasse *X* nennt man den Ausdruck *X::m* einen *qualifizierten Namen*. Er hat den gleichen Typ wie *m* und ist nur dann ein L-Wert, wenn auch *m* ein L-Wert ist. Auf das gewünschte Element kann damit auch dann zugegriffen werden, wenn der Klassenname *X* durch einen anderen Namen verdeckt ist.

Es ist nicht möglich, in einem anderen Programmteil als der Klassendeklaration selbst, Elemente einer Klasse hinzuzufügen. Die Klasse *X* kann nicht Objekte derselben Klasse *X* enthalten, wohl aber Zeiger oder Referenzen auf Objekte derselben Klasse *X* (ähnlich wie bei C-Strukturen und -Varianten).

Geschachtelte Typen

In der Version 2.1 von C++ gehören »Tag«- und **typedef**-Namen, die innerhalb einer Klasse definiert werden, lexikalisch zu dem Gültigkeitsbereich dieser Klasse. Auf derartige Namen kann generell nur über die Notation `xxx::yyy` zugegriffen werden – außer der aktuelle Gültigkeitsbereich ist der der einschließenden Klasse.

Eine Klasse, die innerhalb einer anderen Klasse deklariert wird, heißt *geschachtelte Klasse*. Ihr Name ist lokal zu der sie einschließenden Klasse; die geschachtelte Klasse liegt im Gültigkeitsbereich der sie einschließenden Klasse. Diese Tatsache kann man als rein lexikalische Schachtelung bezeichnen. Die geschachtelte Klasse besitzt keinerlei zusätzliche Privilegien für den Zugriff auf Elemente der sie einschließenden Klasse (dies gilt umgekehrt auch für die einschließende Klasse).

Klassen können auf diese Art und Weise bis zu einer beliebigen Tiefe geschachtelt werden. Ein Beispiel:

```
struct outer
{
   typedef int t;   // 'outer::t' ist ein typedef-Name

   struct inner     // 'outer::inner' ist eine Klasse
   {
```

```
      static int x;

   };
   static int x
      int f();
};
int outer::x;          // Definiert statisches Datenelement
int outer::f()
{
   t x;                // 't' hier direkt sichtbar
   return x;
}
int outer::inner::x;   // Definiert statisches Datenelement
outer::t x  ;          // Hier muß 'outer::t' verwendet werden
```

In der Version 2.0 von C++ gehören »Tag«- und **typedef**-Namen, die innerhalb einer Klasse deklariert werden, zu dem globalen (Datei-)Gültigkeitsbereich, zum Beispiel:

```
struct foo
{
   enum bar ( x );     // 2.0 Regeln: 'bar' gehört zu dem Datei-
                       // Gültigkeitsbereich
                       // 2.1 Regeln: 'bar' gehört zu dem 'foo'-
                       // Gültigkeitsbereich
};
bar x;
```

Der obige Programmtext wird fehlerfrei compiliert.

Da der Programmtext entsprechend der Regeln für Version 2.1 nicht korrekt ist, wird folgende Warnung ausgegeben:

```
Warning: Use qualified name to access nested type 'foo::bar'
```

Element-Zugriffskontrolle

Elemente einer Klasse erhalten Zugriffsattribute entweder standardmäßig (abhängig von Klassenschlüsseln und dem Ort der Deklaration) oder durch einen der drei folgenden Zugriff-Spezifizierer: **public**, **private** und **protected**. Die Bedeutung dieser Attribute ist wie folgt:

public: Das Element kann von jeder Funktion benutzt werden.

private: Das Element kann nur von Elementfunktionen und Friends der Klasse, in der es deklariert wurde, benutzt werden.

protected: Es gilt das gleiche wie für **private**. Zusätzlich ist jedoch die Benutzung durch Elementfunktionen und Friends von Klassen, die von der deklarierten Klasse *abgeleitet* sind, möglich. Das letztere gilt allerdings nur für Objekte vom abgeleiteten Typ. (Abgeleitete Klassen werden im nächsten Abschnitt besprochen.)

Hinweis: **friend**-Funktionsdeklarationen werden von den Zugriff-Spezifizierern nicht beeinflußt (vgl. den Abschnitt »Friends von Klassen« weiter hinten).

Elemente einer Klasse sind standardmäßig **private**. Sie müssen explizit **public** oder **protected** verwenden, um die Voreinstellung zu überschreiben.

Elemente einer Struktur sind standardmäßig **public**, Sie können die Standardeinstellung aber mit **private** oder **protected** ändern.

Elemente einer Variante sind standardmäßig **public**; diese Einstellung kann nicht geändert werden. Die Angabe eines Zugriff-Spezifizierers ist in Zusammenhang mit Varianten-Elementen nicht zulässig.

Ein Zugriff-Spezifizierer bleibt solange für alle folgenden Elementdeklarationen gültig, bis wieder ein Zugriff-Spezifizierer eingesetzt wird. Ein Beispiel:

```
class X {
   int i;     // X::i ist standardmäßig private
   char ch;   // das gilt auch für X::ch
public:
   int j;     // die beiden nächsten sind public
   int k;
protected:
   int l;     // X::l ist protected
};

struct Y {
   int i;     // Y::i ist standardmäßig public
private:
   int j;     // Y::j ist private
public:
   int k;     // Y::k ist public
};

union Z {
   int i;     // standardmäßig public; keine andere Möglichkeit
   double d;
};
```

Die Zugriff-Spezifizierer können in beliebiger Reihenfolge aufgelistet und gruppiert werden. Sie können sich z.B. etwas Schreibarbeit sparen, indem Sie alle **private**-Elemente zusammen gruppieren.

Der Zugriff auf Basisklassen und abgeleitete Klassen

Bei der Deklaration einer abgeleiteten Klasse *D* werden die Basisklassen *B1*, *B2*, ... mit Komma getrennt in einer *Basisliste* aufgeführt:

Klassenschlüssel **D** *: Basisliste* { *<Elementliste>* }

Da eine Basisklasse selbst eine abgeleitete Klasse sein kann, können die Zugriffsattribute rekursiv angewendet werden: Man geht zurück, bis die erste Basisklasse erreicht ist. *D* erbt alle Elemente dieser Basisklassen. (Neu definierte

Basisklassen-Elemente werden auch geerbt, der Zugriff auf sie kann nötigenfalls durch Änderung des Gültigkeitsbereichs erfolgen.) *D* kann nur die **public**- und **protected**-Elemente seiner Basisklasse benutzen. Welche Zugriffsattribute erhalten nun aber die geerbten Elemente, von *D* aus gesehen? Angenommen, *D* will ein **public**-Element aus einer Basisklasse benutzen, es für Funktionen außerhalb aber als **private**-Element schützen. Die Lösung liegt im Gebrauch der Zugriff-Spezifizierer in der *Basisliste*.

Bei der Deklaration von *D* können Sie die Zugriff-Spezifizierer **public**, **protected** oder **private** vor den Klassen in der Basisliste einsetzen:

```
class D : public B1, private B2, ... {
...
}
```

Hinweis: Varianten können keine Basisklassen enthalten und dürfen nicht als Basisklassen verwendet werden.

Diese Modifizierer ändern nicht die Zugriffsattribute der Basiselemente (von der Basisklasse aus gesehen), *können* aber die Zugriffsattribute der Basiselemente, von der abgeleiteten Klasse aus gesehen, ändern.

Ist *D* eine **class**-Deklaration, wird standardmäßig der Zugriff-Spezifizierer **private** verwendet. Dagegen ist **public** der Standard, wenn *D* eine **struct**-Deklaration ist. Die abgeleitete Klasse erbt ihre Zugriffsattribute von einer Basisklasse wie folgt:

- **public**-Ableitung: **public**-Elemente der Basisklasse werden **public**-Elemente der Basisklasse. **protected**-Elemente der Basisklasse werden **protected**-Elemente der abgeleiteten Klasse. **private**-Elemente der Basisklasse bleiben **private**-Elemente der Basisklasse.

- **protected**-Ableitung: Sowohl **public**- als auch **protected**-Elemente der Basisklasse werden **protected**-Elemente der abgeleiteten Klasse. **private**-Elemente der Basisklasse bleiben **private**-Elemente der Basisklasse.

- **private**-Ableitung: Sowohl **public**- als auch **protected**-Elemente der Basisklasse werden **private**-Elemente der abgeleiteten Klasse. **private**-Elemente der Basisklasse bleiben **private**-Elemente der Basisklasse.

In beiden Fällen können die **private**-Elemente der Basisklasse von den Elementfunktionen der abgeleiteten Klasse nicht benutzt werden. Dies gilt allerdings nicht, wenn **friend**-Deklarationen in der Basisklasse explizit einen Zugriff erlauben. Das folgende Beispiel soll dies verdeutlichen:

```
class X : A {              // private A ist der Standard für class
...
}

// class X ist von class A abgeleitet

class Y : B, public C {    // überschreibt den Standard für C
...
}
```

```
// class Y ist durch Mehrfachvererbung von B und C abgeleitet,
// der Standard ist private B

struct S : D {            // Standard für struct ist public D
...                       // struct S ist von D abgeleitet
}

struct T : private D, E {  // überschreibt den Standard für D
                           // E ist standardmäßig public
...
}

// struct T ist von D und E abgeleitet (Mehrfachvererbung),
// der Standard ist public E
```

Der Effekt von Zugriff-Spezifizierern in der Basisliste kann durch Verwendung von qualifizierten Namen in der **public**- oder **protected**-Deklaration der abgeleiteten Klasse noch einmal abgeändert werden:

```
class B {

   int a;                 // standardmäßig private
public:
   int b, c;
   int Bfunc(void);
};
class X : private B {     // a, b, c und Bfunc sind nun private in X
   int d;                 // standardmäßig private, auf a kann
                          // in X nicht zugegriffen werden
public:
   B::c;                  // c war private, jetzt ist es public
   int e;
   int Xfunc(void);
};

int Efunc(X& x);          // extern zu B und X
```

Die Funktion *Efunc* kann nur die **public**-Namen *c*, *e* und *Xfunc* verwenden. Die Funktion *Xfunc* ist ein Element der Klasse *X*, die von **private** *B* abgeleitet ist. *Xfunc* hat demnach Zugriff auf:

- *c*, dessen Zugriffsattribut auf **public** abgeändert ist;
- die Elemente *b* und *Bfunc* von *B*, deren Zugriffsattribute auf **private**-für-*X* abgeändert sind;
- die Elemente *d*, *e* und *Xfunc*, die der Klasse *X* eigen sind.

Xfunc kann nicht auf das Element *a* zugreifen, das **private**-für-*B* ist.

Virtuelle Basisklassen

Eine Basisklasse kann nicht mehrfach in einer abgeleiteten Klasse verwendet werden:

```
class B { ... };
class D : B, B { ... };                // Illegal
```

Allerdings ist es möglich, eine Basisklasse indirekt mehrmals an eine abgeleitete Klasse zu übergeben:

```
class X : public B { ... }
class Y : public B { ... }
class Z : public X, public Y { ... }   // OK
```

In diesem Fall hat jedes Objekt der Klasse *Z* zwei Unterobjekte der Klasse *B*. Falls dies zu Problemen führt, kann das Schlüsselwort **virtual** zum Basisklassen-Spezifizierer hinzugefügt werden:

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

B ist nun eine virtuelle Basisklasse und die Klasse *Z* hat nur ein Unterobjekt der Klasse *B*.

Friends von Klassen

Ein **friend** *F* der Klasse *X* ist eine Funktion oder Klasse, die – obwohl kein Element der Klasse *X* – volle Zugriffsrechte zu den **private**- und **protected**-Elementen von *X* hat. In Bezug auf Gültigkeitsbereich, Deklaration und Definition wird *F* wie eine normale Funktion behandelt.

Da *F* kein Element von *X* ist, liegt es nicht im Gültigkeitsbereich von *X* und kann deshalb nicht mit den Selektionsoperatoren *x.F* bzw. *xptr->F* aufgerufen werden (wobei *x* ein Objekt von *X* und *xptr* ein Zeiger auf ein Objekt von *X* ist).

Um eine Funktion zum Friend einer Klasse zu machen, deklariert man sie in der Klasse mit dem Schlüsselwort **friend**.

Friend-Funktionen, die innerhalb einer Klasse *definiert* sind, gehorchen denselben Inline-Regeln, wie Elementfunktionen (siehe »Inline-Funktionen« weiter oben). Friend-Funktionen werden weder durch die Stellung innerhalb einer Klasse noch durch irgendwelche Zugriff-Spezifizierer beeinflußt. Beispiel:

```
class X {
   int i;                  // private zu X
   friend void friend_func(X*, int);
// friend_func ist nicht private, obwohl sie im
// private-Abschnitt deklariert wird
```

```
public:
     void member_func(int);
};

// Definitionen; beide Funktionen greifen auf private int i zu
void friend_func(X* xptr, int a) { xptr->i = a; }
void X::member_func(int a) { i = a; }

X xobj;

// Man beachte den Unterschied in den Funktionsaufrufen
friend_func(&xobj, 6);
xobj.member_func(6);
```

Alle Funktionen der Klasse *Y* können mit einer einzigen Deklaration zu Friends der Klasse *X* gemacht werden:

```
class Y;                  // unvollständige Deklaration
class X {
   friend Y;
   int i;
   void member_funcX();
};
class Y; {                // die Deklaration wird vervollständigt
   void friend_X1(X&);
   void friend_X2(X*);
   ...
};
```

Die in *Y* deklarierten Funktionen sind Friends von *X*, obwohl sie keinen **friend**-Spezifizierer haben. Sie haben Zugriff auf die **private**-Elemente von *X*, wie z.B. *i* oder *member_funcX*.

Es ist außerdem möglich, daß eine einzelne Elementfunktion der Klasse *X* ein Friend der Klasse *Y* wird:

```
class X {
   ...
   void member_funcX();
}
class Y {
   int i;
   friend void X::member_funcX();
   ...
};
```

Klassen-»Freundschaften« sind nicht transitiv: Ist z.B. *X* Friend von *Y* und *Y* Friend von *Z*, heißt das nicht, daß *X* Friend von *Z* sein muß. »Freundschaften« werden aber vererbt.

Konstruktoren und Destruktoren

Einige spezielle Elementfunktionen bestimmen, wie die Objekte einer Klasse erzeugt, initialisiert, kopiert und gelöscht werden. Konstruktoren und Destruktoren sind die wichtigsten dieser Funktionen. Sie verfügen über viele Eigenschaften der normalen Elementfunktionen. Beispielsweise werden sie innerhalb einer Klasse deklariert und definiert – oder innerhalb einer Klasse deklariert und außerhalb definiert. Sie haben allerdings noch einige besondere Eigenschaften:

1. Sie haben keine Rückgabewert-Deklarationen (auch nicht **void**).

2. Sie können nicht vererbt werden; allerdings kann eine abgeleitete Klasse die Konstruktoren und Destruktoren der Basisklasse aufrufen.

3. Konstruktoren können – wie die meisten C++-Funktionen – Parameterlisten mit Vorgabewerten haben oder die Initialisierungslisten der Elemente benutzen.

4. Destruktoren können **virtual** sein, Konstruktoren nicht.

5. Ihre Adressen können nicht bestimmt werden:

```
int main(void)
{
   ...
   void *ptr = base::base;    // Illegal
   ...
}
```

6. Konstruktoren und Destruktoren können von Turbo C++ generiert werden, wenn sie nicht explizit vom Benutzer definiert wurden. Sie werden außerdem bei vielen Gelegenheiten im Programm aufgerufen, ohne daß im Quelltext ein expliziter Aufruf vorkommt. Jeder Konstruktor oder Destruktor, der vom Compiler generiert wird, ist **public**.

7. Konstruktoren können nicht wie eine normale Funktion aufgerufen werden. Dagegen ist der Aufruf von Destruktoren möglich, wenn ihr voller qualifizierter Name verwendet wird.

```
{
...
   X *p;
...
   p->X::~X():          // Legaler Aufruf des Destruktors
   X::X();              // Illegaler Aufruf des Konstruktors
...
}
```

8. Bei der Definition bzw. dem Löschen von Objekten ruft der Compiler automatisch Konstruktoren und Destruktoren auf.

9. Konstruktoren und Destruktoren können implizit die Operatoren **new** und **delete** aufrufen, wenn eine Speicherplatzreservierung für ein Objekt nötig ist.

10. Ein Objekt, das einen Konstruktor oder Destruktor enthält, kann nicht als Element einer Variante verwendet werden.

Hat eine Klasse *X* einen oder mehrere Konstruktoren, wird bei jeder Definition eines Objekts vom Typ *x* der Klasse *X* einer von ihnen aufgerufen. Der Konstruktor erzeugt und initialisiert *x*. Destruktoren machen diesen Vorgang wieder rückgängig, indem sie das von den Konstruktoren erzeugte Objekt der Klasse löschen.

Konstruktoren werden auch bei der Erzeugung eines lokalen oder temporären Objekts einer Klasse aufgerufen. Destruktoren werden aufgerufen, wenn deren Gültigkeitsbereich verlassen wird.

Konstruktoren

Konstruktoren unterscheiden sich von allen anderen Elementfunktionen dadurch, daß sie den Namen der Klasse tragen, zu der sie gehören. Wenn ein Objekt dieser Klasse erzeugt oder kopiert wird, wird der entsprechende Konstruktor implizit aufgerufen.

Konstruktoren für globale Variablen werden vor dem Aufruf der Funktion *main* aufgerufen. Wird die Direktive *#pragma startup* zur Installation einer Funktion vor *main* verwendet, werden die Konstruktoren für globale Variablen vor dieser Funktion aufgerufen.

Lokale Objekte werden erzeugt, wenn der Gültigkeitsbereich der Variablen aktiv wird. Der Aufruf eines Konstruktors erfolgt auch dann, wenn ein temporäres Objekt der Klasse erzeugt wird.

```
class X
{
public:
   X();      // Konstruktor der Klasse X
};
```

Ein Konstruktor der Klasse *X* kann nicht *X* als Argument haben:

```
class X {
...
public:
   X(X);     // Illegal
}
```

Die Übergabeparameter für einen Konstruktor können jeden Typ haben, außer dem der Klasse, deren Element er ist. Der Konstruktor kann eine Referenz auf seine eigene Klasse als Parameter erhalten. In diesem Fall wird er *Kopierkonstruktor* genannt. Ein Konstruktor, an den keine Argumente übergeben werden, heißt *Standardkonstruktor*.

Der Standardkonstruktor

An den Standardkonstruktor für die Klasse *X* werden keine Argumente übergeben; er hat normalerweise die Form *X::X()*. Wenn kein benutzerdefinierter Konstruktor für eine Klasse existiert, erzeugt Turbo C++ einen Standardkonstruktor. Bei einer Deklaration eines Objekts innerhalb einer Klasse, z.B. *X x*, erzeugt der Standardkonstruktor das Objekt *x*.

Wichtig: Wie alle Funktionen können Konstruktoren Parameter mit Vorgabewerten haben. Beispielsweise akzeptiert der Konstruktor

```
X::X(int, int = 0);
```

ein oder zwei Argumente. Falls das zweite Argument beim Aufruf fehlt, wird 0 eingesetzt. Ähnlich akzeptiert der Konstruktor

```
X::X(int = 5, int = 6);
```

ein, zwei oder auch kein Argument. Der Standardkonstruktor *X::X()* akzeptiert *kein* Argument und darf nicht mit *X::X(int = 0)* verwechselt werden, der eines oder – als Standardkonstruktor – kein Argument akzeptiert.

Bitte vermeiden Sie daher bei der Verwendung von Konstruktoren zweideutige Konstruktionen. Im folgenden Fall können der Standardkonstruktor und der Konstruktor, dem ein Integer übergeben wird, nicht klar auseinander gehalten werden:

```
class X
{
public:
   X();
   X(int i = 0);
};
main()
{
   X one(10);      // OK; verwendet X::X(int)
   X two;          // Illegal; nicht klar, ob X::X() oder
                   // X::X(int = 0) aufgerufen werden soll
...
   return 0;
}
```

Der Kopierkonstruktor

Ein Kopierkonstruktor für die Klasse *X* kann mit einem einzelnen Argument vom Typ *X: X::X(const X&)* oder *X::X(const X&, int = 0)* aufgerufen werden. In einem Kopierkonstruktor sind auch Parameter mit Vorgabewerten erlaubt.

Kopierkonstruktoren werden zum Kopieren eines Klassenobjekts eingesetzt – zum Beispiel wenn Sie ein Objekt mit Initialisierung durch ein anderes Objekt

deklarieren: *X x =y*. Turbo C++erzeugt einen Kopierkonstruktor für die Klasse *X*, wenn einer gebraucht wird und in der Klasse keiner definiert wurde.

Überladen von Konstruktoren

Konstruktoren können überladen werden. Dadurch ist die Erzeugung von Objekten möglich, die von den Initialisierungswerten abhängig sind.

```
class X
{
   int    integer_part;
   double double_part;
public:
   X(int i)     { integer_part = i; }
   X(double d)  { double_part = d; }
};
main()
{
   X one(10);   // ruft X::X(int) auf und
                // setzt integer_part auf 10
   X two(3.14); // ruft X::X(double) auf und
                // setzt double_part auf 3.14
...
  return 0;
}
```

Aufrufreihenfolge von Konstruktoren

Hat eine Klasse eine oder mehrere Basisklassen, werden die Basisklassen-Konstruktoren vor dem Konstruktor der abgeleiteten Klasse aufgerufen. Die Basisklassen-Konstruktoren werden in der Reihenfolge ihrer Deklaration aufgerufen. Beispielsweise werden hier

```
class Y {...};
class X : public Y {...};
X one;
```

die Konstruktoren in der folgenden Reihenfolge aufgerufen:

```
Y();   // Basisklassen-Konstruktor
X();   // abgeleiteter Klassenkonstruktor
```

Gibt es mehrere Basisklassen,

```
class X : public Y, public Z;
X one;
```

werden die Konstruktoren in der Reihenfolge ihrer Deklaration aufgerufen:

```
Y();   // Basisklassen-Konstruktoren kommen zuerst
Z();
X();
```

Konstruktoren für virtuelle Basisklassen werden vor denen für nicht-virtuelle Basisklassen aufgerufen. Sind mehrere virtuelle Basisklassen vorhanden, werden die zugehörigen Konstruktoren in der Reihenfolge ihrer Deklaration aufgerufen. Danach werden die Konstruktoren der nicht-virtuellen Basisklassen aufgerufen, darauf folgen die der abgeleiteten Klasse.

Wurde eine virtuelle Klasse von einer nicht-virtuelllen Basisklasse abgeleitet, dann wird diese zuerst behandelt, um sicherzustellen, daß der Konstruktor der virtuellen Klasse richtig arbeitet. Der Programmteil

```
class X : public Y, virtual public Z;
X one;
```

produziert diese Reihenfolge:

```
Z();        // Initialisierung der virtuellen Basisklasse
Y();        // nicht-virtuelle Basisklasse
X();        // abgeleitete Klasse
```

Ein etwas komplexeres Beispiel:

```
class base;
class base2;
class level1 : public base2, virtual public base;
class level2 : public base2, virtual public base;
class toplevel : public level1, virtual public level2;
toplevel view;
```

Die Abarbeitungsreihenfolge der Konstruktoren für *view* ist wie folgt definiert:

```
base();     // virtuelle Basisklasse ist die höchste in der
            // Hierarchie CR   base wird nur einmal initialisiert
base2();    // die nicht-virtuelle Basis der virtuellen Basis level2
            // muß zur Initialisierung von level2 aufgerufen werden
level2();   // virtuelle Basisklasse
base2();    // nicht-virtuelle Basis von level1
level1();   // weitere nicht-virtuelle Basis
toplevel();
```

Falls eine Klassenhierarchie mehrere Instanzen derselben virtuellen Basisklasse enthält, wird der Konstruktor dieser Basisklasse nur einmal aufgerufen. Existieren jedoch virtuelle und nicht-virtuelle Instanzen der Basisklasse, wird der Klassenkonstruktor einmal für alle virtuellen Instanzen und dann jeweils einmal für jede nicht-virtuellen Instanz der Basisklasse aufgerufen.

Konstruktoren für Elemente eines Array werden in aufsteigender Reihenfolge des Index aufgerufen.

Die Initialisierung von Klassen

Ein Objekt einer Klasse, das nur **public**-Elemente und keine Konstruktoren oder Basisklassen enthält (dies ist normalerweise eine Struktur), kann mit einer Initialisierer-Liste initialisiert werden. Hat eine Klasse einen Konstruktor, müssen ihre Objekte entweder initialisiert werden oder einen Standardkonstruktor haben. Der Standardkonstruktor wird für Objekte genutzt, die nicht explizit initialisiert werden.

Objekte von Klassen mit Konstruktoren können mit einer Liste von Ausdrücken in Klammern initialisiert werden. Diese Liste wird als Argumentliste für den Konstruktor genutzt.

Eine Alternative dazu ist das Gleichheitszeichen (=), gefolgt von einem einzelnen Wert. Dieser Wert muß vom Typ des ersten Arguments sein, das ein Konstruktor dieser Klasse erhält. In diesem Fall gibt es keine weiteren Argumente, oder die restlichen Argumente haben Vorgabewerte. Als Initialisierungswert kann auch ein Objekt des Klassentyps verwendet werden.

Im ersten Fall wird der zugehörige Konstruktor aufgerufen, um das Objekt zu erzeugen. Im zweiten Fall wird zur Initialisierung des Objekts der Kopierkonstruktor benutzt.

```
class X
{
   int i;
public:
   X();           // Die Funktionsrümpfe werden der
   X(int x);      // übersichtlichkeit halber weggelassen
   X(const X&);
};

main()

{
   X one;         // Standardkonstruktor wird aufgerufen
   X two(1);      // Konstruktor X::X(int) wird verwendet
   X three = 1;   // Ruft X::X(int) auf
   X four = one;  // Ruft X::X(const X&) zum Kopieren auf
   X five(two);   // Ruft X::X(const X&)
}
```

Der Konstruktor kann seinen Elementen auf zweierlei Weise Elemente zuweisen. Er kann die Werte als Parameter übernehmen und sie den Elementvariablen innerhalb der Funktion des Konstruktors zuweisen:

```
class X

{
   int a, b;
public:
   X(int i, int j) { a = i; b = j }
};
```

Er kann aber auch eine Initialisierer-Liste benutzen, die vor dem Funktionsrumpf steht:

```
class X
{
   int a, b;
public:
   X(int i, int j) : a(i), b(j) {}
};
```

In beiden Fällen weist eine Initialisierung von *X x(1, 2) x::a* den Wert 1 und *x::b* den Wert 2 zu. Die zweite Methode – die Initialisierer-Liste – ist ein Mechanismus zur Übergabe von Werten an Basisklassen-Konstruktoren.

Basisklassen-Konstruktoren müssen als **public** oder als **protected** deklariert sein, damit sie von einer abgeleiteten Klasse genutzt werden können.

```
class base1
{
   int x;
public:
   base1(int i) { x = i; }
};

class base2
{
   int x;
public:
   base2(int i) : x(i) {}
};

class top : public base1, public base2
{
   int a, b;
public:
   top(int i, int j) : base1(i*5), base2(j+i), a(i) { b = j; }
};
```

In dieser Klassenhierarchie würde eine Deklaration von *top one(1, 2)* die Initialisierung von *base1* mit dem Wert 5 und von *base2* mit dem Wert 3 bedeuten.

Die Methoden der Initialisierung können gemischt werden.

Wie bereits oben erwähnt, werden die Basisklassen in der Reihenfolge ihrer Deklaration initialisiert. Anschließend folgt die Initialisierung der Elemente (ebenfalls in der Deklarationsreihenfolge), unabhängig von der Initialisierungsliste.

```
class X
{
   int a, b;
public:
  X(int i, j) : a(i), b(a+j) {}
};
```

In dieser Klasse hat die Deklaration von *X x(1,1)* eine Zuweisung von 1 zu *x::a* und 2 zu *x::b* zur Folge. Basisklassen-Konstruktoren werden vor der Konstruktion der abgeleiteten Klassenelemente aufgerufen.

Die Werte der abgeleiteten Klasse können nicht geändert werden und sich dann noch auf die Erzeugung der Basisklasse auswirken.

```
class base
{
   int x;
public:
   base(int i) : x(i) {}
};

class derived : base
{
   int a;
public:
   derived(int i) : a(i*10), base (a) { }  // Vorsicht! Der Basis
                                           // wird ein uninitiali-
                                           // siertes a übergeben
};
```

Bei dieser Klassenkonstruktion führt der Aufruf eines abgeleiteten *d(1)* nicht zum Wert für das Basisklassen-Element *x*. Der an den Basisklassen-Konstruktor übergebene Wert ist undefiniert.

Wenn Sie für einen nicht-**inline**-Konstruktor eine Initialisierer-Liste verwenden wollen, plazieren Sie die Liste *nicht* in die Klassendefinition, sondern an die Stelle, an der die Funktion definiert wird.

```
derived::derived(int i) : a(i)
{
...
}
```

Destruktoren

Der Destruktor einer Klasse wird aufgerufen, um Elemente eines Objekts freizugeben, bevor das Objekt selbst gelöscht wird. Der Destruktor einer Elementfunktion trägt den Namen der Klasse mit einer führenden Tilde (~). Ein Destruktor akzeptiert weder Argumente noch liefert er einen Rückgabetyp oder -wert.

```
class X
{
public:
   ~X();    // Destruktor für die Klasse X
};
```

Ist kein Destruktor explizit definiert, wird einer vom Compiler erzeugt.

Wann werden Destruktoren aufgerufen?

Ein Destruktor wird implizit aufgerufen, wenn eine Variable ihren deklarierten Gültigkeitsbereich verläßt. Destruktoren für lokale Variablen werden aufgerufen, wenn der Block, in dem sie deklariert sind, nicht mehr aktiv ist. Bei globalen Variablen werden Destruktoren als Teil der Exit-Prozedur nach *main* aufgerufen.

Wenn Zeiger auf Objekte ungültig werden, wird *nicht* implizit ein Destruktor aufgerufen. Zum Löschen eines derartiges Objekts muß demnach der **delete**-Operator aufgerufen werden.

Destruktoren werden in der umgekehrten Reihenfolge wie die zugehörigen Konstruktoren aufgerufen (vgl. den Abschnitt »Aufrufreihenfolge von Konstruktoren« weiter vorne in diesem Kapitel).

atexit, #pragma exit und Destruktoren

Globale Objekte sind aktiv, bis der Code in allen Exit-Prozeduren ausgeführt wurde. Lokale Variablen (auch die in *main* deklarierten), werden gelöscht, wenn sie ungültig werden. Die Ausführungsreihenfolge am Ende eines Turbo C++ Programms ist folgende:

- *atexit*-Funktionen werden in der Reihenfolge ausgeführt, in der sie aufgeführt wurden.
- #*pragma exit*-Funktionen werden in der Reihenfolge ihrer Prioritätsnummern ausgeführt.
- Die Destruktoren für globale Variablen werden aufgerufen.

exit und Destruktoren

Wird die Prozedur *exit* innerhalb eines Programms aufgerufen, wird für keine lokale Variable im aktuellen Gültigkeitsbereich ein Destruktor aufgerufen.

Globale Variablen werden in der Reihenfolge ihrer Deklaration gelöscht.

abort und Destruktoren

Wird die *abort*-Routine in einem Programm aufgerufen, werden keine Destruktoren aktiviert. Dies gilt auch für Variablen mit globalem Gültigkeitsbereich.

Ein Destruktor kann auch explizit auf eine der folgenden Arten aufgerufen werden: indirekt durch einen Aufruf des Operators **delete** oder direkt durch die

Angabe des vollen qualifizierten Namens des Destruktors. Mit **delete** können auch Objekte gelöscht werden, die mit **new** erzeugt wurden.

Explizite Aufrufe von Destruktoren sind nur für Objekte nötig, die durch Aufrufe von **new** spezifische Adressen erhalten haben.

```
class X {
...
   ~X();
...
};

void* operator new(size_t size, void *ptr)
{
   return ptr;
}

char buffer[sizeof(X)];
main()
{
   X* pointer = new X;
   X* exact_pointer;
   exact_pointer = new(&buffer) X; // Der Zeiger wird mit der Adresse
                                   // des Puffers initialisiert
...
   delete pointer;                 // Mit delete wird das Objekt, auf
                                   // das Pointer zeigt, gelöscht
   exact_pointer->X::~X();         // direkter Aufruf für
                                   // die Speicherfreigabe
```

Virtuelle Destruktoren

Ein Destruktor kann als **virtual** deklariert werden. Dies erlaubt es einem Zeiger auf ein Basisklassen-Objekt, den richtigen Destruktor aufzurufen, wenn der Zeiger aktuell auf ein abgeleitetes Klassenobjekt zeigt. Der Destruktor einer Klasse, die von einer Klasse mit einem virtuellen Destruktor abgeleitet ist, ist selbst virtuell.

```
class color
{
public:
   virtual ~color();           // Virtueller Destruktor für color
};

class red : public color
}
public:
   ~red();                     // Der Destruktor für red ist virtuell
};
```

```
class brightred : public red
{
public:
   ~brightred();          // Der Destruktor für brightred
                          // ist ebenfalls virtuell
};
```

Die oben aufgelisteten Klassen und die folgenden Deklarationen

```
color *palette[3];
palette[0] = new red;
palette[1] = new brightred;
palette[2] = new color;
```

liefern folgende Resultate:

```
delete palette[0];
// Der Destruktor für red wird aufgerufen, gefolgt von dem
// Destruktor für color.

delete palette[1];
// Der Destruktor für brightred wird aufgerufen, gefolgt von
// dem für red und color.

delete palette[2];
// Der Destruktor für color wird aufgerufen.
```

Wurden keine Destruktoren als virtuell deklariert, rufen **delete** *palette*[0], **delete** *palette*[1] und **delete** *palette*[2] nur den Destruktor für die Klasse *color* auf. Dadurch würden aber die ersten beiden Elemente, die tatsächlich vom Typ *red* und *brightred* sind, nicht auf korrekte Weise gelöscht.

Überladene Operatoren

In C++ können die Aktionen der meisten Operatoren so umdefiniert (*überladen*) werden, daß sie bei der Verwendung mit Objekten einer bestimmten Klasse spezielle Funktionen ausführen. Wie bei allen überladenen C++-Funktionen bestimmt auch hier der Compiler, welche Funktion er aufrufen soll, durch die Anzahl und den Typ der Argumente.

Alle in Kapitel 2 »Semantische Grammatik« im Abschnitt »Operatoren-Semantik« aufgeführten Operatoren können überladen werden, ausgenommen folgende:

```
.   .*   ::   ?:
```

Die Präprozessor-Symbole # und ## können ebenfalls nicht überladen werden.

Das Schlüsselwort **operator**, gefolgt von dem Operatorsymbol, wird *Operator-Funktionsname* genannt. Er wird wie ein normaler Funktionsname verwendet, wenn die neue (überladene) Aktion des Operators definiert wird.

Ein mit Argumenten aufgerufener Funktionsoperator verhält sich wie ein Operator, der seine Operanden in einem Ausdruck bearbeitet. Die Operatorfunktion kann weder die Anzahl der Argumente noch die Abarbeitungsreihenfolge oder die Assoziativität ändern (vergleichen Sie Tabelle 2.10). Dazu ein Beispiel:

```
class complex {
   double real, imag;                // standardmäßig private
public:
   ...
   complex() { real = imag = 0; }    // inline-Konstruktor
   complex(double r, double i = 0) {  // noch ein Konstruktor
        real = r; imag = i;
   }
   ...
}
```

Hinweis: Diese Klasse dient nur zu Demonstrationszwecken. Sie ist nicht mit der Klasse *complex* in der Laufzeitbibliothek identisch.

Wir könnten eine Funktion definieren, die zwei komplexe Zahlen addiert:

```
complex AddComplex(complex c1, complex c2);
```

Es wäre aber natürlicher, wenn wir statt

```
c3 = AddComplex(c1, c2);
```

folgendes schreiben könnten:

```
complex c1(0,1), c2(1,0), c3;
c3 = c1 + c2;
```

Der Operator + kann überladen werden, indem man die folgende Deklaration in die Klasse *complex* einfügt

```
friend complex operator +(complex c1, complex c2);
```

und folgendes (evtl. **inline**) definiert:

```
complex operator +(complex c1, complex c2)
{
   return complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

Operator-funktionen

Operatorfunktionen können zwar direkt aufgerufen werden, der Aufruf erfolgt aber für gewöhnlich indirekt durch Benutzung des überladenen Operators:

```
c3 = c1.operator + (c2);    // entspricht  c3 = c1 + c2
```

Abgesehen von **new** und **delete**, die ihre eigenen Regeln haben, muß eine Operatorfunktion eine nicht-statische Elementfunktion sein oder mindestens

ein Argument vom Klassentyp haben. Die Operatorfunktionen =, ( ), [ ] und -> müssen nicht-statische Elementfunktionen sein.

Die »Vererbung« von überladenen Operatoren

Mit Ausnahme des Zuweisungs-Operators =() (vgl. den Abschnitt »Überladen des Zuweisungsoperators =« weiter hinten in diesem Kapitel) werden alle überladenen Operatorfunktionen für die Klasse *X* an die von *X* abgeleiteten Klassen vererbt. Dabei gelten die gleichen Auflösungsregeln wie für überladene Funktionen. Wenn *X* eine Basisklasse für *Y* ist, kann eine überladene Operatorfunktion für *X* möglicherweise nochmals für *Y* überladen werden.

Überladen von new und delete

Die Operatoren **new** und **delete** können ebenfalls überladen werden, um alternative Routinen für die Speicherverwaltung zur Verfügung zu stellen. Ein benutzerdefinierter Operator **new** muß **void*** zurückliefern und *size_t* als ersten Parameter haben (der Typ *size_t* ist in STDLIB.H definiert). Ein benutzerdefinierter Operator **delete** muß einen Rückgabewert vom Typ **void** und **void*** als erstes Argument haben; ein zweites Argument vom Typ *size_t* ist optional.

Dazu ein Beispiel:

```
#include <stdlib.h>

class X {
   ...
public:
   void* operator new(size_t size) { return newalloc(size);}
   void operator delete(void* p) { newfree(p); }
   X() { /* hier initialisieren */ }
   X(char ch) { /* hier ebenfalls */ }

   ~X() { /* hier aufräumen */ }
   ...
};
```

Das Argument *size* gibt die Größe des zu erzeugenden Objekts an. *newalloc* und *newfree* sind benutzerdefinierte Funktionen für die Reservierung und die Freigabe von Speicherplatz. Konstruktor- und Destruktor-Aufrufe von Objekten der Klasse *X* (oder Objekten von Klassen, die von *X* abgeleitet sind, und nicht über eigene überladene **new**- und **delete**-Operatoren verfügen) rufen die entsprechenden benutzerdefinierten Operatorfunktionen *X::operator new*() bzw. *X::operator delete*() auf.

Die Operatorfunktionen *X::operator new* und *X::operator delete* sind statische Elemente von *X* (gleichgültig, ob sie explizit als **static** definiert sind oder nicht) und können deshalb keine virtuellen Funktionen sein.

Die vordefinierten (globalen) Standardoperatoren **new** und **delete** können weiterhin im Gültigkeitsbereich von *X* verwendet werden. Dies ist explizit mit dem globalen Zugriffsoperator (*::operator new* und *::operator delete*) möglich oder implizit beim Erzeugen und Löschen von Klassenobjekten, die nicht zu *X* gehören oder von *X* abgeleitet sind. Sie können die Standardoperatoren **new** und **delete** z.B. bei der Definition von überladenen Versionen verwenden:

```
void* X::operator new(size_t s)
{
   void* ptr = new char[s];   // Aufruf von Standard-new
   ...
   return ptr;
}

void X::operator delete(void* ptr)
{
   ...
   delete (void*) ptr;        // Aufruf von Standard-delete
}
```

Das Argument *size* wird verwendet, da die von *X* abgeleiteten Klassen den *X::operator new* erben. Die Größe eines abgeleiteten Klassenobjekts kann sich von derjenigen der Basisklasse unterscheiden.

Das Überladen von unären Operatoren

Ein unärer Präfix- oder Suffix-Operator kann auf zwei Arten überladen werden: durch die Deklarierung einer nicht-statischen Elementfunktion ohne Argumente oder durch die Deklarierung einer Nicht-Elementfunktion mit einem Argument. Wenn @ für einen unären Operator steht, können *@x* und *x@* sowohl als *x.operator@()* als auch als *operator@(x)* interpretiert werden (abhängig von den vorhandenen Deklarationen). Wurden beide Formen deklariert, werden Zweideutigkeiten durch den standardmäßigen Vergleich der Argumente aufgelöst.

In der Version 2.0 von C++ werden die überladenen Operatoren ++ oder -- sowohl als Präfix- als auch als Suffix-Operatoren verwendet, z.B.:

```
struct foo
{
   operator++();
   operator--();
} x;
void func()
{
   x++;      // Ruft x.operator++() auf
   ++x;      // Ruft x.operator++() auf
   x--;      // Ruft x.operator--() auf
   --x;      // Ruft x.operator--() auf
}
```

Wenn in C++ 2.1 *operator++* oder *operator--* als Elementfunktionen ohne Parameter definiert sind oder als Nicht-Elementfunktionen mit einem Argument, wird nur der Präfix-Operator ++ oder -- überladen. Einen Suffix-Operator ++ oder -- können Sie nur durch die Definition einer Elementfunktion mit einem **int**-Argument oder einer Nicht-Elementfunktion mit einem Klassen-Argument und einem **int**-Argument überladen. Dazu fügen Sie folgende Zeilen zu dem obigen Programmtext hinzu:

```
operator++(int);
operator--(int);
```

Wenn nur die Präfix-Version von operator++ oder operator-- überladen ist und der Operator für ein Objekt als Suffix-Operator verwendet wird, gibt der Compiler eine Warnung aus und ruft dann den Präfix-Operator auf, so daß Programmtext der Version 2.0 compiliert wird. In diesem Fall werden folgende Warnungen ausgegeben:

```
Warning: Overloaded prefix 'operator ++' used as a postfix
operator in function func()

Warning: Overloaded prefix 'operator --' used as a postfix
operator in function func()
```

Das Überladen von binären Operatoren

Ein binärer Operator kann ebenfalls auf zweierlei Arten überladen werden: Durch die Deklarierung einer nicht-statischen Elementfunktion mit einem Argument oder durch die Deklarierung einer Nicht-Elementfunktion (normalerweise als **friend**) mit zwei Argumenten. Wenn @ einen binären Operator repräsentiert, kann $x@y$ entweder als $x.operator@(y)$ oder als $operator@(x,y)$ interpretiert werden (abhängig von den vorhandenen Deklarationen). Wurden beide Formen deklariert, werden Zweideutigkeiten durch den standardmäßigen Vergleich der Argumente aufgelöst.

Überladen des Zuweisungsoperators =

Der Zuweisungsoperator = kann nur durch die Deklarierung einer nichtstatischen Elementfunktion überladen werden. Zum Beispiel:

```
class String {
   ...
   String& operator = (String& str);
   ...
   String (String&);
   ~String();
}
```

Mit den entsprechenden Definitionen von *String::operator =*() ermöglicht dieses Programm die in anderen Sprachen üblichen Stringzuweisungen der Form *str1 = str2*. Im Gegensatz zu anderen Operatorfunktionen kann die Zuweisungsoperator-Funktion von abgeleiteten Klassen nicht geerbt werden. Gibt es für eine beliebige Klasse *X* keinen benutzerdefinierten Operator =, wird = standardmäßig als Element-zu-Element-Zuweisung der Elemente von *X* definiert:

```
X& X::operator = (const X& source)
{
   // elementweise Zuordnung
}
```

Überladen des Funktionsaufruf-Operators ()

Der Funktionsaufruf

Primärer-Ausdruck <Ausdrucksliste>

wird als binärer Operator mit den Operanden *Primärer-Ausdruck* und *Ausdrucksliste* (kann leer sein) interpretiert. Die entsprechende Operatorfunktion ist **operator()**. Diese Funktion kann vom Benutzer nur mit Hilfe einer nicht-statischen Elementfunktion für eine Klasse *X* (und für abgeleitete Klassen) definiert werden. Der Aufruf *x(arg1,arg2)* wird als *x.operator()(arg1,arg2)* interpretiert, wobei *x* ein Objekt der Klasse *X* ist.

Überladen des Index-Operators []

Die Index-Operation

Primärer-Ausdruck Ausdruck

wird als binärer Operator mit den Operanden *Primärer-Ausdruck* und *Ausdruck* interpretiert. Die entsprechende Operatorfunktion ist **operator[]**; sie kann vom Benutzer nur mit Hilfe einer nicht-statischen Elementfunktion für eine Klasse *X* (und für abgeleitete Klassen) definiert werden. Der Ausdruck *x[y]* wird als *x.operator[](y)* interpretiert, wobei *x* ein Objekt der Klasse *X* ist.

Überladen des Operators ->

Der Zugriff auf Klassenelemente mit

Primärer-Ausdruck ->Ausdruck

wird als unärer Operator interpretiert. Die Funktion **operator->** muß eine nicht-statische Elementfunktion sein.

Der Ausdruck *x->m* wird als (*x.operator->*())->*m* interpretiert, wobei *x* ein Objekt der Klasse *X* ist. Die Funktion *operator->*() muß daher entweder einen Zeiger auf ein Klassenobjekt oder ein Objekt einer Klasse, für die **operator->** definiert ist, zurückliefern.

Virtuelle Funktionen

Mit Hilfe von virtuellen Funktionen können abgeleitete Klassen für sich spezifische Versionen einer Basisklassen-Funktion bereitstellen. Eine virtuelle Funktion wird in der Basisklasse mit dem Schlüsselwort **virtual** deklariert und kann dann in jeder abgeleiteten Klasse neu definiert werden (virtuelle Funktionen müssen Elementfunktionen sein). Wenn Anzahl und Typ der Parameter sowie der Rückgabetyp übereinstimmen, *überschreibt* die neue Funktion die Basisklassen-Funktion.

Die obigen Bedingungen sind zwingend: Es ist generell in C++nicht möglich, zwei verschiedene Funktionen zu definieren, die sich nur im Rückgabetyp unterscheiden. Hat die überschreibende Funktion denselben Namen, aber eine andere Parameterliste als die virtuelle Basisklassen-Funktion, behandelt C++die beiden als unterschiedliche Funktionen – die Funktionen sind weder überschrieben noch überladen (weil sie nicht denselben Gültigkeitsbereich haben) und die Basisklassen-Funktion wird von der neuen Funktion verdeckt.

Enthält die Klasse *B* eine virtuelle Funktion *vf* und die von *B* abgeleitete Klasse *D* eine Funktion *vf* des gleichen Typs, dann lautet – bei einem Aufruf von *vf* für ein Objekt *d* von *D* – der Aufruf *D::vf*. Dies gilt wie gesagt auch dann, wenn der Zugriff über einen Zeiger oder eine Referenz auf *B* erfolgt. Dazu ein Beispiel:

```
struct B {
   virtual void vf1 ();
   virtual void vf2 ();
   virtual void vf3 ();
   void f();
};

class D : public B {
   virtual void vf1();   // virtual ist hier legal, aber redundant
   void vf2(int);        // nicht virtuell, da vf2 eine
                         // andere Parameterliste benutzt
   char vf3();           // Illegal: nur der Rückgabetyp hat
                         // sich geändert!
   void f();
};

void extf()
{
   D d;                  // Deklaration eines D-Objekts
   B* bp = &d;           // Standardkonvertierung von D* zu B*
   bp->vf1();            // ruft D::vf1 auf
```

```
    bp->vf2();          // ruft B::vf2 auf, da vf2 von D andere
                        // Argumente benötigt
    bp->f();            // ruft B::f auf (f ist nicht virtuell)
}
```

Die überschreibende Funktion *vf1* in *D* ist automatisch virtuell. Der Spezifizierer **virtual** darf mit einer überschreibenden Funktionsdeklaration in der abgeleiteten Klasse verwendet werden, seine Verwendung ist allerdings redundant. Die Interpretation eines virtuellen Funktionsaufrufs hängt vom Typ des Objekts ab, für das der Aufruf erfolgt. Bei nicht-virtuellen Funktionsaufrufen ist die Interpretation nur vom Typ des Zeigers oder der Referenz abhängig, der bzw. die das betreffende Objekt bezeichnet.

Virtuelle Funktionen müssen Elemente einer Klasse sein, dürfen aber nicht statisch sein. Eine virtuelle Funktion kann ein **friend** einer anderen Klasse sein.

Eine virtuelle Funktion in der Basisklasse muß – wie alle Elementfunktionen einer Basisklasse – entweder definiert oder als abstrakt deklariert sein:

```
class B {
virtual void vf(int) = 0;    // = 0 bedeutet `abstrakt'
```

In einer von einer derartigen Basisklasse abgeleiteten Klasse muß jede abstrakte Funktion als abstrakt definiert oder neu deklariert sein (vgl. den nächsten Abschnitt »Abstrakte Klassen«).

Ist eine virtuelle Funktion in der Basis definiert, muß sie nicht notwendigerweise in der abgeleiteten Klasse neu definiert werden. Bei einem Aufruf wird dann einfach auf die Basisfunktion zugegriffen.

Virtuelle Funktionen zahlen aber einen Preis für ihre Vielseitigkeit: Jedes Objekt in der abgeleiteten Klasse braucht einen Zeiger auf eine Tabelle mit virtuellen Funktionen, damit zur Laufzeit die richtige Funktion gewählt wird (späte Bindung).

Abstrakte Klassen

Eine *abstrakte Klasse* ist eine Klasse mit mindestens einer abstrakten Funktion. Die Spezifizierung einer Funktion als abstrakt erfolgt mit dem Spezifizierer `= 0`.

Abstrakte Klassen können nur als Basisklassen für andere Klassen verwendet werden. Es ist nicht möglich, Objekte einer abstrakten Klasse zu generieren. Eine abstrakte Klasse kann nicht als Parametertyp oder als Funktionsrückgabetyp verwendet werden. Es ist jedoch möglich, Zeiger auf eine abstrakte Klasse zu deklarieren. Wenn zur Initialisierung kein temporäres Objekt benötigt wird, sind Referenzen auf eine abstrakte Klasse erlaubt. Dazu ein Beispiel:

```
class shape {       // abstrakte Klasse
   point center;
   ...
public:
   where() { return center; }
```

```
    move(point p) { center = p; draw(); }
    virtual void rotate(int) = 0;   // abstrakte Funktionen
    virtual void draw() = 0;        // abstrakte Funktionen
    virtual void hilite() = 0;      // abstrakte Funktionen
    ...
};
shape x;          // Fehler: versuchte Erzeugung eines Objekts
                  // einer abstrakten Klasse
    shape* sptr;  // Zeiger auf abstrakte Klasse ist zulässig
    shape f();    // Fehler abstrakte Klasse kann kein
                  // Rückgabetyp sein
int g(shape s);   // Fehler: abstrakte Klasse kann kein
                  // Funktionsparametertyp sein
shape& h(shape&); // Referenz auf abstrakte Klasse als Rück-
                  // gabewert oder Funktionsargument ist OK
```

Nehmen wir an, *D* ist eine abgeleitete Klasse mit der abstrakten Klasse *B* als unmittelbarer Basisklasse, die eine abstrakte Funktion *pvf* enthält. Stellt *D* keine Definition für *pvf* bereit, wird *pvf* eine abstrakte Elementfunktion von *D*, und *D* wird dadurch zu einer abstrakten Klasse. Mit der oben eingeführten Klasse *shape* sieht das folgendermaßen aus:

```
class circle : public shape {  // circle abgeleitet von
                               // abstrakter Klasse, bleibt abstrakt
    int radius;                // private
public:
    void rotate(int)  {  }     // definiert virtuelle Funktion:
                               // zur Rotation eines Kreises wird
                               // keine Aktion benötigt
    void draw();               // circle::draw muß anderswo
                               // definiert werden
};
```

Elementfunktionen können von einem Konstruktor einer abstrakten Klasse aufgerufen werden. Der direkte oder indirekte Aufruf einer abstrakten Funktion durch einen derartigen Konstruktor führt jedoch zu einem Laufzeitfehler.

Der Gültigkeitsbereich in C++

Die lexikalischen Gültigkeitsbereichsregeln für C++, mit Ausnahme des Klassen-Gültigkeitsbereichs, folgen den generellen C-Regeln, mit dem Unterschied, daß C++, anders als C, Daten- und Funktionsdeklarationen an jeder Stelle zuläßt, an der auch Anweisungen möglich sind. Diese Flexibilität erfordert aber Vorsicht bei der Interpretation der Begriffe »einschließender Gültigkeitsbereich« und »Deklarationszeitpunkt«.

Gültigkeitsbereich Klasse

Der Name *M* eines Elements der Klasse *X* hat den Gültigkeitsbereich Klasse, (»lokal zu *X*«) und kann nur in folgenden Fällen verwendet werden:

- in Elementfunktionen von *X*;
- in Ausdrücken wie *x.M*, wobei *x* ein Objekt von *X* ist;
- in Ausdrücken wie *xptr->M*, wobei *xptr* ein Zeiger auf ein Objekt von *X* ist;
- in Ausdrücken wie *X::M* oder *D::M*, wobei *D* eine von *X* abgeleitete Klasse ist;
- in vorgreifenden Referenzen innerhalb der Klasse, in der *M* ein Element ist.

Namen von Funktionen, die als **friend** von *X* deklariert sind, sind keine Elemente von *X*; ihre Namen haben nur einschließenden Gültigkeitsbereich.

Verbergen

Ein Name kann durch eine explizite Deklaration desselben Namens in einem einschließenden Block oder einer Klasse verborgen werden. Mit dem Zugriffsoperator und dem Klassennamen kann aber immer noch auf ein verborgenes Klassenelement zugegriffen werden: *X::M*. Ein verborgener (globaler) Name mit Dateigültigkeitsbereich kann mit dem unären Operator `::` referenziert werden, z.B. `::g`. Ein Klassenname *X* kann durch den Namen eines Objekts, einer Funktion oder eines Aufzählungstyps innerhalb des Gültigkeitsbereichs von *X* verborgen werden, unabhängig von der Reihenfolge, in der die Namen deklariert sind. Auf den verborgenen Klassennamen kann durch Voranstellen des geeigneten Schlüsselwortes weiterhin zugegriffen werden: **class**, **struct** oder **union**. Der Deklarationspunkt für einen Namen *x* liegt unmittelbar hinter der vollständigen Deklaration, aber vor dem Initialisierer (falls vorhanden).

Zusammenfassung der C++ Gültigkeitsbereichsregeln

Die folgenden Regeln betreffen alle Namen, einschließlich der **typedef**- und Klassennamen, vorausgesetzt, daß C++ derartige Namen im genannten Zusammenhang erlaubt.

1. Der Name selbst wird auf Mehrdeutigkeit hin überprüft. Werden keine Mehrdeutigkeiten innerhalb des Gültigkeitsbereiches festgestellt, wird die Zugriffssequenz eingeleitet.

2. Wenn kein Zugriffsfehler auftritt, wird der Typ des Objekts, der Funktion, der Klasse, von **typedef**, etc. überprüft.

3. Wenn der Name außerhalb der Funktion oder Klasse verwendet wird oder den unären Zugriffsoperator `::` als Präfix hat, *und* wenn der Name nicht durch den binären Operator `::` oder die Elementauswahloperatoren `.` und `->` qualifi-

ziert ist, dann muß der Name ein globales Objekt, eine globale Funktion oder einen globalen Aufzählungstyp bezeichnen.

4. Erscheint der Name *n* in der Form *X::n*, *x.n* (wobei *x* eine Instanz von *X* oder eine Referenz auf *X* ist) oder *ptr->n* (wobei *ptr* ein Zeiger auf *X* ist), dann ist *n* der Name für ein Element von *X* oder ein Element einer Basisklasse von *X*.

5. Jeder Name, auf den die bisherigen Beschreibungen nicht zutreffen und der in einer statischen Elementfunktion verwendet wird, muß in dem Block, in den er gehört, oder in einem einschließenden Block deklariert werden, oder er muß ein globaler Name sein. Die Deklaration eines lokalen Namens *n* verbirgt Deklarationen von *n* in einschließenden Blöcken und in globalen Deklarationen von *n*. Namen in unterschiedlichen Gültigkeitsbereichen werden nicht überladen.

6. Jeder Name, auf den die bisherigen Beschreibungen nicht zutreffen und der in einer nicht-statischen Elementfunktion der Klasse *X* verwendet wird, muß in dem Block, in den er gehört, oder in einem einschließenden Block deklariert werden, oder er muß ein Element der Klasse *X* bzw. einer Basisklasse von *X* oder ein globaler Name sein. Die Deklaration eines lokalen Namens *n* verbirgt Deklarationen von *n* in einschließenden Blöcken und in globalen Deklarationen von *n*. Wird *n* in einer Elementfunktion einer Klasse deklariert, verbirgt diese Deklaration Deklarationen von *n* in Elementen der Klasse. Die Deklaration eines Elementnamens verbirgt Deklarationen desselben Namens in Basisklassen.

7. Der Name eines Funktionsparameters in einer Funktionsdefinition liegt im Gültigkeitsbereich des äußersten Blocks der Funktion. Der Name eines Funktionsparameters in einer nicht-definierenden Funktionsdeklaration hat überhaupt keinen Gültigkeitsbereich. Der Gültigkeitsbereich eines Parameters mit Vorgabewert wird durch den Deklarationspunkt des Parameters bestimmt, Ausdrücke für den Vorgabewert können aber nicht auf lokale Variable oder nicht-statische Klassenelemente zugreifen. Vorgabewerte werden bei jeder Aufrufposition ausgewertet.

8. Ein Konstruktor-Initialisierer (vgl. *ctor-Initialisierer* in der Klassendeklaratorsyntax, Tabelle 2.39 wird im Gültigkeitsbereich des äußersten Blocks des Konstruktors ausgewertet, so daß er sich auf die Namen der Konstruktorparameter beziehen kann.

Templates

Mit Templates oder *generischen* bzw. *parametrisierten Typen* können Sie eine Familie gleichartiger Funktionen oder Klassen aufbauen.

Im vorliegenden Abschnitt stellen wir Ihnen das Konzept und anschließend einige Spezifika vor.

Die Syntax von Templates sieht folgendermaßen aus:

Template-Deklaration
Template <Template-Argumentliste >Deklaration

Template-Argumentliste
  Template-Argument
  Template-Argumentliste, Template-Argument

Template-Argument
  Typ-Argument
  Argument-Deklaration

Typ-Argument
  class-*Bezeichner*

Template-Klassenname
  Template-Name <Template-Argumentliste >

Template-Argumentliste
  Template-Argument
  Template-Argumentliste, Template-Argument

Template-Argument
  Ausdruck
  Typ-Name

Funktions-Templates

Sehen wir uns die Funktion *max*(*x*,*y*) an, die das größere der beiden Argumente liefert. *x* und *y* können von einem beliebigen sortierbaren Typ sein. Da Turbo C++ eine streng auf Typen ausgerichtete Sprache ist, müssen die Argumenttypen zum Zeitpunkt der Compilierung deklariert werden. Ohne Templates sind viele überladene Versionen von *max*() erforderlich – eine Version für jeden unterstützten Datentyp, obwohl der Programmtext für jede Version im Prinzip identisch ist. Jede Version vergleicht die Argumente miteinander und liefert das größere Argument zurück. Sehen Sie sich dazu das folgende Beispiel an:

```
int max(int x, int y)
{
   return (x> y) ? x : y;
}
long max(long x, long y)
{
   return (x > y) ? x : y
}
   .
   .
   .
```

gefolgt von weiteren *max*-Versionen. Zur Lösung dieses Problems kann die Verwendung eines Makros dienen:

```
#define max(x,y) ((x > x) ? x : y)
```

Die Verwendung von **#define** umgeht allerdings die Typenüberprüfung, die ein wesentlicher Vorteil von C++ gegenüber C ist. Der Einsatz von Makros ist in C++ häufig überflüssig. Die Aufgabe von *max(x,y)* liegt im Vergleich von kompatiblen Typen. Das Makro seinerseits erlaubt – ungünstigerweise – auch einen Vergleich von **int**- und **struct**-Typen, die nicht kompatibel sind.

Ein weiteres Problem bei der Verwendung des Makros liegt darin, daß eine Ersetzung ausgeführt wird, wo Sie es gar nicht wünschen:

```
class foo
{
   public:
     int max(int, int);  // Syntaxfehler; der Makro-Ausdruck
                         // würde eingesetzt!
};
```

Bei Verwendung eines Template können Sie eine Schablone für gleichartige überladene Funktionen definieren, indem Sie den Datentyp selbst als Parameter verwenden. Definition eines Funktion-Templates:

```
template <class T>
T max(T x, T y)
{
   return (x> y) ? x : y;
};
```

Der Datentyp wird durch den Template-Parameter *<class T>* repräsentiert. Beim Einsatz in einer Anwendung generiert der Compiler die entsprechende Funktion und zwar gemäß des im Aufruf verwendeten Datentyps:

```
int i;
Myclass a, b;

int j = max(i,0);        // Integer-Argumente
Myclass m = max(a,b)     // Argumente vom Typ Myclass
```

Für *<class T>* kann jeder beliebige Typ (aber keine Klasse) verwendet werden. Der Compiler ruft dann den entsprechenden *operator>()* auf, so daß Sie *max* mit Argumenten jedes beliebigen Typs, für den der Größer-als-Operator definiert ist, einsetzen können.

Eine Template-Funktion überschreiben

Das vorhergehende Beispiel wird *Funktions-Template* (oder generische Funktion) genannt. Eine spezielle Instantiierung eines Funktions-Template heißt *Template-Funktion*. Sie können eine Template-Funktion für einen speziellen Typ mit einer Nicht-Template-Funktion überschreiben:

```
#include <string.h>

char *max(char *x, char *y)
{
   return(strcmp(x, y)>0) ?x:y;
}
```

Wenn Sie die Funktion mit String-Argumenten aufrufen, wird sie anstelle der automatischen Template-Funktion ausgeführt. In diesem Fall verhindert der Aufruf der Funktion einen sinnlosen Vergleich zwischen zwei Zeigern.

Mit Template-Funktionen, die vom Compiler generiert werden, werden nur einfache Argument-Konvertierungen ausgeführt.

Die Argumenttypen einer Template-Funktion müssen alle formalen Argumente des Template abdecken. Wenn dies nicht erfolgt, gibt es keine Möglichkeit, beim Aufruf der Funktion die tatsächlichen Werte des nicht verwendeten Template-Arguments zu ermitteln.

Implizite und explizite Template-Funktionen

Bei der Auflösung von Aufrufen überladender Funktionen, also der Suche nach der »passendsten« Variante, ignoriert der Compiler Template-Funktionen, die implizit von ihm selbst erzeugt wurden.

```
template<class T> T max(T a, T b)
{
      return (a > b) ? a : b;
}

void   f(int i, char c)
{
      max(i, i);      // Ruft max(int, int) auf
      max(c, c);      // Ruft max(char, char) auf
      max(i, c);      // Keine übereinstimmung für max(int,char)
      max(c, i);      // Keine übereinstimmung für max(char,int)
```

Dieser Programmtext gibt folgende Fehlermeldungen aus:

```
Could not find match for 'max(int,char)' in function f(int,char)
Could not find match for 'max(char,int)' in function f(int,char)
```

Wenn der Anwender eine Template-Funktion aber explizit deklariert, wird die Funktion vollständig in die Suche nach der besten Übereinstimmung einbezogen:

```
template< class T> T max(T a, T b)
{
      return (a>b) ? a: b;
}

int    max(int,int);   // Deklariert max(in,int) explizit
{
      max(i, i);      // Ruft max(int,int) auf
      max(c, c);      // Ruft max(char,char) auf
      max(i, c);      // Ruft max(int,int) auf
      max(c, i);      // Ruft max(int,int) auf
}
```

Klassen-Templates

Mit einem Klassen-Template (auch *generische Klasse* oder *Klassengenerator*) können Sie ein Muster für Klassendefinitionen anlegen. Generische Containerklassen sind dafür ein Beispiel. Sehen Sie sich das folgende Beispiel einer Vektorklasse an (ein eindimensionales Array).

Die grundlegenden Operationen (Einfügen, Löschen, Indizieren, etc.), die für den Typ ausgeführt werden, sind immer dieselben, unabhängig davon, ob Sie einen Vektor mit Integerwerten oder mit einem anderen Typ haben. Der Elementtyp wird als *Typ*-Parameter behandelt und das System erzeugt typensichere Klassendefinitionen.

Definition von Klassen-Templates:

```
#include <iostream.h>

template <class T>
class Vector
{
   T *data;
   int size;

public:
   Vector(int)
   ~Vector() {delete[] data;}
   T& operator[] (int i) {return data[i];}
};

     // Beachten Sie die Syntax für Definitionen außerhalb
     // der Klasse:
template <class T>
Vector<T>::Vector(int n)
{
   data = new T[n];
   size = n;
};

main()
{
   Vector<int> x(5);  // Generiert einen Vektor mit Integern
   for (int = 0; i < 5; ++i)
      x[i] = i;
   for (i = 0; i < 5; ++i);
      cout << x[i] << ' ';
   cout << '\n';
   return 0;
}
     // Ausgabe: 0 1 2 3 4
```

Wie bei Funktions-Templates kann eine *explizite* Template-Klassen-Definition die automatische Definition eines Typs überschreiben:

```
class Vector<char *> { ... }
```

Das Symbol *Vector* muß immer den Datentyp in spitzen Klammern bei sich haben. Es kann nicht alleine stehen, ausgenommen sind einige Fälle in der ursprünglichen Template-Definition.

Eine vollständige Implementierung einer Vektorklasse finden Sie in der Datei vectimp.h im Quelltext der Container-Klassenbibliothek im Unterverzeichnis \BORLANDC\CLASSLIB\INCLUDE.

Argumente

Die obigen Beispiele verwenden alle nur ein Template-Argument. Es sind aber mehrere Argumente erlaubt. Template-Argumente können auch zusätzlich zu den Datentypen Werte repräsentieren:

```
template<class T, int size = 64 > class Buffer { ... }
```

Argumente, die kein Template-Typ sind, wie beispielsweise *size*, können Voreinstellungen haben. Der Typ, der für ein Nicht-Template-Argument angegeben werden kann, muß ein konstanter Ausdruck sein:

```
const int N = 128;
int i = 256;

Buffer<int, 2*N>b1;    // OK
Buffer<float, i>b2;    // Fehler: i ist keine Konstante
```

Da jede Instanz eines Klassen-Template eine Klasse ist, erhält sie ihre eigene Kopie der statischen Elemente, so wie Template-Funktionen ihre eigene Kopie der statischen lokalen Variablen erhalten.

Hinweis: Im Ausdruck *<class T>* muß *T* nicht immer eine Klasse sein.

Spitze Klammern

Geben Sie acht bei der Verwendung der rechten spitzen Klammer bei der Instantiierung:

```
Buffer<char, (x > 100 ? 1024 : 64)> buf;
```

Im obigen Beispiel würde die spitze Klammer zwischen *x* und *100* die Template-Argumentliste schließen, wenn das zweite Argument nicht in runden Klammern stünde.

Typensichere generische Listen

Wenn Sie viel nahezu identischen Programmtext schreiben müssen, sollten Sie immer an Templates denken. Bei der folgenden Klassendefinition, einer generischen Listenklasse,

```
class GList
{
   public:
      void insert( void * );
      void *peek();
        // ...
};
```

liegt das Problem darin, daß sie nicht typensicher ist. Üblicherweise werden zur Lösung dieses Problems wiederholte Klassendefinitionen verwendet. Da für die Einfügungen keine Typenüberprüfung durchgeführt wird, haben Sie keinen Überblick darüber, was zurückgeliefert wird. Sie können dieses Problem mit Hilfe einer umschließenden Klasse beheben:

```
class FooList : public GList
{
public:
   void insert( Foo *f ) { GList::insert( f ); }
   Foo *peek() { return (Foo*)GList::peek(); }
       // ...
};
```

Damit ist die Typensicherheit hergestellt. *Insert* kann nur Argumente vom Typ Zeiger-auf-*Foo* oder Objekt-abgeleitet-von-*Foo* haben. Dadurch speichert der zugrundeliegende Container Zeiger, die nur auf Objekte vom Typ *Foo* zeigen. Das bedeutet, daß die Typumwandlung (cast) *FooList::peek* immer sicher ist und Sie eine korrekte *FooList* erzeugen. Um dies nun auch für *BarList*, *BazList*, etc. zu erreichen, müssen Sie weitere Klassendefinition wiederholen. Um dieses Problem der wiederholten Klassendefinitionen zur Gewährleistung der Typensicherheit zu lösen, verwenden Sie ein Template:

```
template <class T> class List : public GList;
{
public:
   void insert( T *t )  { GList::insert( t ); }
   T *peek()  { return (T *)GList::peek(); }
   // ...
};

   List<Foo> fList;  // Erzeugt eine Klasse für eineFooList und eine
                     // Instanz namens fList.
   List<Bar> bList;  // Erzeugt eine Klasse für eineBarList und eine
                     // Instanz namens bList.
   List<Baz> zList;  // Erzeugt eine Klasse für eineBazList und eine
                     // Instanz namens zList.
```

Durch die Verwendung von Templates können Sie beliebige typensichere Listen mit einer einfachen Deklaration anlegen. Dabei wird durch die Typkonvertierung von der umschließenden Klasse kein Programmtext erzeugt, so daß während der Laufzeit kein zusätzlicher Aufwand wegen dieser Typensicherheit entsteht.

Objekte statt Zeiger

Eine weitere Technik ist das Einbinden von Objekten durch Eliminieren von Zeigern. Dadurch kann die Anzahl der erforderlichen Aufrufe von virtuellen Funktionen reduziert werden, weil der Compiler die Typen der Objekte kennt. Dies ist ein großer Vorteil, wenn die virtuellen Funktionen klein genug sind, um als Inline-Funktionen verwendet werden zu können. Es ist aber schwierig Inline-Funktionen zu verwenden, wenn sie über Zeiger aufgerufen werden.

In diesem Fall kennt der Compiler die Typen der Objekte nicht, auf die gezeigt wird.

```
template <class T> aBase
{
    // ...
  private:
    T buffer;
};

class anObject : public aSubject, public aBase<aFilebuf>
{
    // ...
};
```

Alle Funktionen in *aBase* können in *aFilebuf* definierte Funktionen direkt aufrufen, ohne dafür einen Zeiger zu benötigen. Wenn eine der in *aFilebuf* vorhandenen Funktionen *inline* verwendet werden kann, erreichen Sie eine Erhöhung der Ausführungsgeschwindigkeit.

4. Der Präprozessor

Obwohl Turbo C++ einen integrierten Single-Pass-Compiler verwendet, der die Übersetzung in einem Durchgang (pass) erledigt, ist es nützlich, die Terminologie früherer Multi-Pass-Compiler beizubehalten.

Bei diesen werden im ersten Durchgang (vom Präprozessor) alle Include-Dateien in den Quelltext einbezogen, Direktiven für die bedingte Compilierung geprüft, Makros erweitert und schließlich eine Zwischendatei erzeugt, die dann für die weiteren Übersetzungsdurchläufe zur Verfügung steht.

Die folgende Erläuterung der Präprozessor-Direktiven, ihrer Syntax und Semantik gilt für die Präprozessor-Funktionen, die in den Compiler von Turbo C++ integriert sind.

Der Turbo C++ Präprozessor enthält einen ausgeklügelten Makro-Prozessor, der Ihren Quelltext durchsucht, bevor der eigentliche Compiler an die Arbeit geht. Er sucht nach Präprozessor-Direktiven (auch Steuerzeilen genannt) und interpretiert die in ihnen enthaltenen Symbole. Der Präprozessor erhöht Ihre Produktivität und Flexibilität beim Programmieren in folgenden Bereichen:

- Definieren von Makros, die den Programmieraufwand verringern und die Lesbarkeit des Quelltextes verbessern. Mit Hilfe einiger Makros kann der Overhead von Funktionsaufrufen ausgeschaltet werden.

- Einbinden von Text aus anderen Dateien (Header-Dateien), die Standardbibliotheksfunktionen, benutzerdefinierte Prototypen sowie manifeste Konstanten enthalten.

- Festlegen von Direktiven für die bedingte Compilierung, um die Portierbarkeit zu verbessern und und Testphasen zu vereinfachen.

Jede Zeile, die mit einem # beginnt, wird als Präprozessor-Direktive behandelt, außer # befindet sich in einer Zeichenkette, in einer Zeichenkonstanten oder ist Teil eines Kommentars. Whitespaces, die dem Zeichen # vorausgehen oder folgen, werden ignoriert (dies gilt nicht für Zeilenvorschübe).

Präprozessor-Direktiven werden normalerweise an den Anfang des Quelltextes geschrieben, könnten aber an jeder beliebigen Stelle im Programm stehen.

Die komplette Syntax für Präprozessor-Direktiven in Turbo C++ finden Sie in Tabelle 4.1.

Tabelle 4.1: Syntax für Präprozessor-Direktiven in Turbo C++

Präprozessor-Datei:
Gruppe

Gruppe:
Gruppenteil
Gruppe Gruppenteil

Gruppenteil:
<Präprozessor-Symbole>Zeilenvorschub
if-Sektion
Steuerzeile

if-Sektion:
if-Gruppe <elif-Gruppen><else-Gruppe>endif-Zeile

if-Gruppe:
#if *Konstantenausdruck Zeilenvorschub <Gruppe>*
#ifdef *Bezeichner Zeilenvorschub <Gruppe>*
#ifndef *Bezeichner Zeilenvorschub <Gruppe>*

elif-Gruppen:
elif-Gruppe
elif-Gruppen elif-Gruppe

elif-Gruppe:
#elif *Konstantenausdruck Zeilenvorschub <Gruppe>*

else-Gruppe:
#else *Zeilenvorschub <Gruppe>*

endif-Zeile:
#endif *Zeilenvorschub*

Steuerzeile:
#include *Präprozessor-Symbole Zeilenvorschub*
#define *Bezeichner Ersetzungsliste Zeilenvorschub*
#define *Bezeichn. LKlammer <Bezeichnerliste>) Ersetzungsliste Zeilenvorschub*
#undef *Bezeichner Zeilenvorschub*
#line *Präprozessor-Symbole Zeilenvorschub*
#error *<Präprozessor-Symbole>Zeilenvorschub*
#pragma *<Präprozessor-Symbole>Zeilenvorschub*
#pragma warn *Aktion Kürzel Zeilenvorschub*
#pragma inline *Zeilenvorschub*
*Zeilenvorschub*

Aktion: eines der folgenden Zeichen:
\+ - .

Kürzel: eine der folgenden Zeichenfolgen:

amb	amp	apt	aus	big	cln
cpt	def	dup	eff	mod	par
pia	pro	rch	ret	rng	rpt

rvl sig str stu stv sus
ucp use voi zst

LKlammer:
eine linke Klammer, die unmittelbar auf das vorangehende Element folgt (ohne vorangehenden Whitespace).

Ersatzliste:
<Präprozessor-Symbole>

Präprozessor-Symbole:
Präprozessor-Symbol
Präprozessor-Symbole Präprozessor-Symbol

Präprozessor-Symbol:
Header-Dateiname (nur für **#include**)
Bezeichner (Schlüsselwörter werden nicht erkannt)
Konstante
String-Literal
Operator
Interpunktionszeichen
jedes Zeichen (ungleich Whitespace), das nicht in die zuvor aufgeführten Klassen fällt.

Header-Dateiname:
<h-Zeichenfolge>

h-Zeichenfolge:
H-Zeichen
h-Zeichenfolge H-Zeichen

H-Zeichen:
jedes Zeichen außer den Steuerzeichen, dem Zeilenvorschub (`\n`), den Zeichen `>` und `"`.

Zeilenvorschub:
das Zeichen newline (`\n`)

Die Null-Direktive

Die Null-Direktive besteht aus einer Zeile, die nur das Zeichen # enthält. Diese Direktive wird immer ignoriert.

Die Direktiven #define und #undef

Die Direktive **#define** definiert ein *Makro*. Makros stellen einen Mechanismus dar, mit dem Symbole durch eine Menge von formalen, funktionsähnlichen Parametern ersetzt werden können.

Einfache #define-Makros

Im einfachsten Fall – ohne Parameter – lautet die Syntax:

#define *Makro_Bezeichner <Symbol_Sequenz>*

Jedes Auftreten von *Makro_Bezeichner* in Ihrem Quelltext nach dieser Steuerzeile wird durch die möglicherweise leere *Symbol_Sequenz* ersetzt werden (es gibt dabei ein paar Ausnahmen, auf die wir später eingehen werden). Diese Ersetzungen nennt man *Makroerweiterungen*. Die *Symbol_Sequenz* nennt man auch Rumpf des Makros.

Erscheint ein Makrobezeichner innerhalb einer Zeichenkette, einer Zeichenkonstanten oder eines Kommentars des Quelltextes, wird das Makro *nicht* erweitert.

Eine leere Symbolsequenz sorgt für die Entfernung des Makrobezeichners aus dem Quelltext:

```
#define HI "Einen schönen Tag noch!"
#define leer
#define NIL ""
...
puts (HI);      // Erweitert zu
                // puts("Einen schönen Tag noch!");
puts (NIL);     // Erweitert zu puts("");
puts ("leer")   // KEINE Erweiterung von leer!
// Auch KEINE Erweiterung von leer in Kommentaren.
```

Nach jeder einzelnen Makroerweiterung wird der erzeugte Text nochmals durchsucht. Dies erlaubt den Einsatz von *geschachtelten Makros*: Der erweiterte Text kann weitere Makrobezeichner enthalten, die ersetzt werden sollen. Wenn das Makro jedoch zu etwas erweitert wird, was wie eine Präprozessor-Direktive aussieht, wird diese vom Präprozessor ignoriert:

```
#define HOLESTD #include <stdio.h>
...
HOLESTD         // Compilerfehler
```

HOLESTD wird erweitert zu *#include <stdio.h>*. Der Präprozessor selbst wird jedoch diese scheinbar gültige Direktive nicht befolgen, sondern sie sofort an den Compiler übergeben. Der Compiler wird `#include <stdio.h>` als ungültige Eingabe zurückweisen. Ein Makro wird nicht während seiner eigenen Erweiterung erweitert. Daher wird *#define A A* nicht endlos erweitert.

Die Direktive #undef

Mit **#undef** können Sie ein Makro außer Kraft setzen:

#undef *Makro_Bezeichner*

Diese Zeile löst jede Verbindung zwischen Makrobezeichner und Symbolsequenz; die Makro-Definition wird vom Präprozessor ab sofort vergessen, der Makrobezeichner ist undefiniert.

Innerhalb der **#undef**-Zeilen finden keine Makroerweiterungen statt.

Der Status *definiert* oder *undefiniert* ist unabhängig von seiner tatsächlichen Definition eine wichtige Eigenschaft eines Bezeichners. Mit den bedingten Direktiven **#ifdef** und **#ifndef** läßt sich nicht nur prüfen, ob ein Bezeichner gegenwärtig definiert ist oder nicht, sie bieten auch einen flexiblen Mechanismus zur Steuerung vieler Aspekte der Compilierung.

Jeder Makrobezeichner, dessen Definition aufgehoben wurde, kann mit **#define** erneut definiert werden, indem die alte oder eine andere Symbolsequenz verwendet wird.

```
#define BLOCK_SIZE 512
...
buff = BLOCK_SIZE*blks; // Erweitert zu 512*blks
...
#undef BLOCK_SIZE
// Verwenden von BLOCK_SIZE wäre jetzt ungültig:
// unbekannter Bezeichner!
...
#define BLOCK_SIZE 128  // erneute Definition
...
buff = BLOCK_SIZE*blks; // ergibt 128*blks
...
```

Der Versuch, einen bereits definierten Makrobezeichner erneut zu definieren, führt zu einer Warnung – außer die Definition ist *exakt* dieselbe. Wenn Definitionen eines Makros in anderen Header-Dateien vorkommen könnten, sollten Sie folgende Strategie anwenden:

```
#ifndef BLOCK_SIZE
   #define BLOCK_SIZE 512
#endif
```

Die mittlere Zeile wird übergangen, wenn BLOCK_SIZE bereits definiert ist, andernfalls wird das Makro entsprechend definiert.

Beachten Sie, daß kein Semikolon (;) notwendig ist, um eine Präprozessor-Direktive zu beenden. Jedes Zeichen in der Symbolsequenz – auch ein Strichpunkt – wird in der Makroerweiterung auftauchen. Die Symbolsequenz endet mit dem ersten Zeilenvorschub-Zeichen, vor dem kein Backslash steht. Jede Folge von Whitespace, Kommentare eingeschlossen, wird jeweils durch ein einzelnes Leerzeichen ersetzt. Assembler-Programmierer müssen der Versuchung widerstehen, folgende Notation zu verwenden:

```
#define BLOCK_SIZE = 512 // ?? Symbolsequenz enthält ein =
```

Die Option Definitionen

Bezeichner können innerhalb der integrierten Entwicklungsumgebung im Dialogfenster *Definitionen* im Menü *Optionen/Compiler/Code-Generierung*) definiert, ihre Definition kann aber nicht aufgehoben werden (vgl. Kapitel 1 »Die integrierte Entwicklungsumgebung« im *Benutzerhandbuch*).

Schlüsselwörter und geschützte Wörter

Es ist zulässig, aber nicht ratsam, Schlüsselwörter von Turbo C++ als Makrobezeichner zu verwenden:

```
#define int long // zulässig, aber möglicherweise katastrophal
#define INT long // zulässig und möglicherweise nützlich
```

Die folgenden vordefinierten globalen Bezeichner dürfen *nicht* unmittelbar auf eine #*define*- oder #*undef*-Direktive folgen:

__STDC__ __DATE__
__FILE__ __TIME__
__LINE__

Beachten Sie die führenden und abschließenden doppelten Unterstriche.

Makros mit Parametern

Makros mit Parametern haben folgende Syntax:

#define *Makro_Bezeichner*(<*Argumente_Liste*>) *Symbol_Sequenz*

Beachten Sie, daß kein Whitespace zwischen dem Makrobezeichner und der linken Klammer (zulässig ist. Die optionale *Argumente_Liste* ist eine Folge von Bezeichnern, die durch Kommas getrennt werden und einer Liste von Argumenten in einer C-Funktion ähneln. Jeder durch ein Komma abgetrennter Bezeichner spielt die Rolle eines *formalen Arguments* oder *Platzhalters*.

Hinweis: Ein Komma innerhalb von runden Klammern in der Argumentliste wird als Teil des Arguments und nicht als Begrenzer des Arguments behandelt.

Derartige Makros werden in dem auf die Makro-Definition folgenden Quelltext aufgerufen durch:

Makro_Bezeichner <*Whitespace*> (<*Aktuelle_Argumente_Liste*>)

Diese Syntax ist also identisch mit der eines Funktionsaufrufs. Tatsächlich sind auch viele »Funktionen« der Laufzeitbibliothek als Makros implementiert. Es gibt jedoch einige wichtige semantische Unterschiede und potentielle Fallstricke, die bei Makroaufrufen beachtet werden sollten. (Wir werden ein paar Seiten später darauf zurückkommen.)

Die optionale *Aktuelle_Argumente_Liste* muß dieselbe Anzahl von durch Kommas getrennten Symbolen (aktuelle Argumente) aufweisen wie die formale *Argumente_Liste* in der #*define*-Zeile: Für jedes formale Argument muß auch ein aktuelles Argument angegeben werden. Falls die Anzahl der Argumente in beiden Listen unterschiedlich ist, wird eine Fehlermeldung ausgegeben.

Ein Makroaufruf bewirkt ein zweimaliges Ersetzen: Zunächst werden der Makrobezeichner und die geklammerten Argumente durch die Symbolsequenz ersetzt. Im zweiten Schritt werden alle formalen Argumente, die in der Symbolsequenz stehen, durch die korrespondierenden tatsächlichen Argumente der *aktuelle_Argumente_Liste* ersetzt. Beispielsweise wird

```
#define CUBE(x) ((x) * (x) * (x))
...
int n,y;
n = CUBE (y);
```

durch folgendes ersetzt:

```
n = ((y)*(y)*(y));
```

Ähnlich wird die letzte Zeile von

```
#define SUM (a,b) ((a) + (b))
...
int i,j,sum;
sum = SUM(i,j);
```

erweitert zu

```
sum = ((i) + (j));
```

Der Grund für die vielen Klammern wird sofort klar, wenn man sich überlegt, was bei folgendem Aufruf

```
n = CUBE(y +1);
```

herauskommen würde, wenn man die inneren Klammern in der Makro-Definition wegließe. Das Makro würde zu *n =y+1*y+1*y+1* erweitert und in folgenden Ausdruck zerlegt werden:

```
n = y +(1*y) +(1*y) +1  // != (y+1)     3 es sei denn y=0 oder y=-3!
```

Genauso wie bei den einfachen Makro-Definitionen erfolgt ein nochmaliges Durchsuchen, um eingebettete Makrobezeichner zu entdecken und gegebenenfalls zu erweitern.

Bei der Verwendung von Makros mit Parametern sollten Sie folgende Punkte beachten:

1. *Geschachtelte Klammerebenen und Kommas:* Die *Aktuelle_Argumente_Liste* darf geschachtelte Klammerebenen enthalten, vergessen Sie dabei nicht die schließenden Klammern; Kommas in Anführungszeichen oder Klammern werden nicht als Argumentbegrenzer behandelt:

```
#define ERRMSG(x, str) showerr("Fehler",x,str)
#define SUM(x,y) ((x) + (y))
```

```
...
ERRMSG(2,"Enter drücken und dann Esc");
// erweitert zu
   showerr("Fehler",2,"Enter drücken und dann Esc");
return SUM(f(i,j), g(k,l));
// erweitert zu return((f(i,j)) + (g(k,l)));
```

2. *Symbole verschmelzen mit ##:* Sie können zwei Symbole miteinander verbinden, indem Sie sie in der Makro-Definition durch ## (plus optionalem Whitespace auf beiden Seiten) verbinden. Der Präprozessor entfernt vorhandenen Whitespace und die ## und kombiniert die zwei separaten Symbole zu einem neuen Symbol. Auf diese Weise lassen sich neue Bezeichner konstruieren. Nach einer Makro-Defintion wie

```
#define VAR(i,j) (i##j)
```

erzeugt der Aufruf *VAR(x,6)* den Text *x6*. Diese Möglichkeit ist ein Ersatz für die ältere – nicht portierbare – Form *(i/**/j)*.

3. *Argumente mit # in Strings konvertieren:* Das Symbol # kann vor ein formales Makroargument gestellt werden, um das aktuelle Argument nach der Erweiterung in einen String umzuwandeln. Nach der folgenden Makrodefinition

```
#define TRACE(flag) printf(#flag "=%d\n", flag)
```

wird das Programmtext-Fragment

```
int grenzwert = 1024;
TRACE(grenzwert);
```

expandiert zu

```
int grenzwert = 1024;
printf("grenzwert" "=%d\n", grenzwert);
```

und behandelt als:

```
int grenzwert = 1024;
printf("grenzwert=%d\n", grenzwert);
```

4. *Zeilenfortsetzung mit dem Backslash:* Eine lange Symbolsequenz kann mit Hilfe des Backslash (\) über mehrere Zeilen ausgedehnt werden. Der Backslash und das anschließende Zeilenvorschub-Zeichen werden bei der Makroerweiterung entfernt:

```
define WARN "Das ist in Wirklichkeit \
eine einzeilige Warnung"
...
puts (WARN);
// Ausgabe: Das ist in Wirklichkeit eine einzeilige Warnung
```

5. *Nebeneffekte und andere Fallstricke*: Wegen der Ähnlichkeit von Funktions- und Makroaufrufen werden oft die kleinen, aber wichtigen Unterschiede vernachlässigt. Bei einem Makroaufruf wird keine automatische Typenüberprüfung durchgeführt, die Nichtübereinstimmung der Datentypen von formalen und

aktuellen Argumenten kann bizarre Resultate hervorbringen, vor denen man nicht sofort gewarnt wird. Entsprechend schwierig ist die Fehlersuche.

Makros können auch zu unerwünschten Nebeneffekten führen, speziell dann, wenn Argumente mehrmals ausgewertet werden. Vergleichen Sie *CUBE* und *cube* in dem folgenden Beispiel:

```
int cube(int x) {
    return x*x*x;
}
#define CUBE(x) ((x)*(x)*(x))
...
int b = 0, a =3;
b = cube(a++);
// cube() wird das aktuelle Argument
// a = 3 übergeben; => b= 27; anschließend wird a auf 4 erhöht
a = 3;
b = CUBE(a++);
// Ergibt ((a++)*(a++)*(a++)); jetzt ist a = 6;
```

Der Endwert von *b* hängt davon ab, was der Compiler mit dem erweiterten Ausdruck anfängt.

Dateien mit #include einbinden

Die Präprozessor-Direktive #*include* fügt andere spezifizierte Dateien, sogenannte *Include-Dateien*, *Header-Dateien* oder kurz *Header* in den Quelltext ein. Die Syntax kennt drei Varianten:

#include *<Header_Name>*
#include *"Header_Name"*
#include *Makro_Bezeichner*

Hinweis: Die spitzen Klammern sind tatsächliche Symbole, keine Metasymbole, die anzeigen, daß *Header_Name* optional ist.

Die dritte Variante setzt voraus, daß erstens weder ein < noch ein " als erstes nicht Whitespace-Zeichen nach dem #*include* erscheint, und daß zweitens eine Makrodefinition existiert, die den *Makro_Bezeichner* in einen gültig begrenzten Header-Namen – *<Header_Name>* oder *"Header_Name"* – erweitert.

Die erste und zweite Variante implizieren, daß keine Makroerweiterung versucht werden soll; das heißt: *Header_Name* wird nicht nach Makrobezeichnern durchsucht, sondern muß einen gültigen Dateinamen in DOS darstellen, der mit einer Dateinamenerweiterung (traditonell ist dies *.h* für Header) angegeben wird und optional um die Pfadangabe erweitert werden kann.

Der Präprozessor entfernt die #*include*-Zeile und ersetzt diese im Quelltext durch den gesamten Text der angegebenen Header-Datei. Die Quelltextdatei wird nicht wirklich geändert, es sieht nur vom Standpunkt des Compilers so aus. Die #*include*-Direktive kann daher Gültigkeitsdauer und -bereich von Bezeich-

nern der eingelesenen Datei beeinflussen. Wenn Sie in *Header_Name* explizit einen Suchpfad angeben, dann wird ausschließlich dieses Verzeichnis durchsucht.

Der Unterschied zwischen *<Header_Name>* und *"Header_Name"* besteht in den angewendeten Algorithmen für das Auffinden der eingebundenen Datei; wir erläutern sie im folgenden Abschnitt.

Header-Dateien suchen mit <Header_Name>

Die Variante *<Header_Name>* definiert eine Standarddatei als Include-Datei; gesucht wird in allen voreingestellten Include-Verzeichnissen und in der Reihenfolge, die Sie definiert haben. Kann keine Datei mit der angegebenen Bezeichnung in den voreingestellten Verzeichnissen gefunden werden, erhalten Sie eine Fehlermeldung.

Header-Dateien suchen mit "Header_Name"

Die Variante *"Header_Name"* bezeichnet eine benutzerdefinierte Include-Datei. Die Datei wird zunächst im aktuellen Verzeichnis (das ist normalerweise das Verzeichnis, in dem auch der zu übersetzende Quelltext steht) gesucht. Wird die Datei dort nicht gefunden, werden alle voreingestellten Include-Verzeichnisse durchsucht. Das folgende Beispiel demonstriert die besprochenen Unterschiede:

```
#include <stdio.h>
// Header im voreingestellten Include-Verzeichnis

#define meininc "C:\TURBOC\INCLUDE\meiner.h"
// Hinweis: Einfache Backslashes sind hier zulässig;
// innerhalb einer C-Anweisung müßten Sie aber:
// "C:\\TURBOC\\INCLUDE\\meiner.h" angeben

#include meininc
// Makroerweiterung

#include "meininc.h"
// Keine Makroerweiterung
```

Nach der Erweiterung bewirkt die zweite #*include*-Direktive, daß der Präprozessor ausschließlich das Verzeichnis C:\TURBOC\INCLUDE\ nach der Datei meiner.h durchsucht. Die dritte #*include*-Direktive bewirkt, daß erst das aktuelle Verzeichnis nach der nicht existierenden meininc.h durchsucht wird und danach die voreingestellten Verzeichnisse.

Bedingte Compilierung

Turbo C++ unterstützt die bedingte Compilierung: bestimmte Zeilen werden durch den Präprozessor vor der Übersetzung des Quelltextes durch leere Zeilen ersetzt. Alle Zeilen, die mit einem # beginnen (außer den Direktiven *#if*, *#ifdef*, *#ifndef*, *#else*, *#elif* und *#endif*) werden ignoriert, genauso wie die Zeilen, die aufgrund von Direktiven nicht übersetzt werden sollen. Alle Direktiven für die bedingte Compilierung müssen in der Datei abgeschlossen werden, in der sie begonnen haben.

Bedingungsdirektiven: #if, #elif, #else und #endif

Die Direktiven zur bedingten Compilierung *#if*, *#elif*, *#else* und *#endif* funktionieren genauso wie normale Bedingungsanweisungen in C. Sie werden wie folgt verwendet:

#if *Konstantenausdruck-1*
<Sektion-1>
***<*#elif** *Konstantenausdruck-2 Zeilenvorschub Sektion-2>*
...
***<*#elif** *Konstantenausdruck-n Zeilenvorschub Sektion-n>*
<*#else<*Zeilenvorschub*>*End-Sektion*>*
#endif
...

Falls *Konstantenausdruck-1* (Gegenstand einer Makroerweiterung) ungleich Null (*True*) ergibt, wird der Code (möglicherweise leer) von *Sektion-1*, bei dem es sich um Präprozessor-Direktiven oder um normale Quelltextzeilen handeln kann, vom Präprozessor verarbeitet und gegebenenfalls dem Compiler übergeben. Falls *Konstantenausdruck-1* mit Null (*False*) bewertet wird, wird *Sektion-1* ignoriert (weder die Makroerweiterung noch die anschließende Übersetzung werden ausgeführt).

Im ersten Fall (*True*) wird, nachdem *Sektion-1* vom Präprozessor verarbeitet wurde, der Quelltext bis zur entsprechenden *#endif*-Direktive übersprungen (die diesen Zweig beendet) und mit dem folgenden Quelltext fortgefahren. Im zweiten Fall (*False*) wird die Kontrolle an die nächste *#elif*-Zeile – falls vorhanden – übergeben, in der der *Konstantenausdruck-2* bewertet wird. Falls dieser *True* ergibt, wird *Sektion-2* bearbeitet und anschließend die Kontrolle an das nächste passende *#endif* übergeben. Liefert *Konstantenausdruck-2* das Ergebnis *False*, geht die Kontrolle an das nächste *#elif* über. Dies geht so weiter, bis entweder *#else* oder *#endif* erreicht wird. Das optionale *#else* wird als Alternativzweig verwendet, wenn alle vorhergehenden Konstantenausdrücke *False* geliefert haben. *#endif* beendet die ganze Bedingungsfolge.

Jede einzelne Sektion kann weitere Bedingungen enthalten, die beliebig tief verschachtelt werden können; jeder mit *#if* geöffnete Zweig muß durch ein entsprechendes *#endif* wieder geschlossen werden.

Das Endergebnis ist: *nur eine* Sektion (möglicherweise eine leere) wird zur weiteren Verarbeitung übergeben. Die übergangenen Sektionen sind nur relevant, um den Weg zu irgendwelchen verschachtelten Bedingungen offen zu halten, damit jedes #*if* mit dem dazugehörigen #*endif* geschlossen werden kann. Die zu prüfenden Konstantenausdrücke müssen einen ganzzahligen Wert liefern.

Der Operator defined

Dieser Operator bietet einen alternativen, flexibleren Weg, um zu prüfen, ob Kombinationen von Bezeichnern definiert sind oder nicht. Er gilt nur innerhalb von #*if*- und #*elif*-Ausdrücken.

Die Ausdrücke **defined**(*Bezeichner*) oder **defined** *Bezeichner* (die Klammern sind optional) liefern das Ergebnis 1 (*True*), wenn das Symbol bereits (mit #*define*) definiert worden ist und die Definition nicht (mit #*undef*) aufgehoben wurde; ansonsten wird der Wert 0 (*False*) geliefert. Die Direktive

```
#if defined(meinsym)
```

ist daher identisch mit

```
#ifdef meinsym
```

Der Vorteil besteht darin, daß man *defined* wiederholt in einem komplexen Ausdruck, der einer #*if*-Direktive folgt, verwenden kann, zum Beispiel

```
#if defined(meinsym) && !defined(deinsym)
```

Die bedingten Direktiven #ifdef und #ifndef

Mit Hilfe dieser bedingten Direktiven können Sie prüfen, ob ein Bezeichner gegenwärtig definiert ist, das heißt, ob eine #*define*-Direktive für diesen Bezeichner existiert und noch gültig ist. Die Zeile

```
#ifdef Bezeichner
```

hat, wenn *Bezeichner* definiert, exakt denselben Effekt wie

```
#if 1
```

und falls er gegenwärtig nicht definiert ist, denselben Effekt wie

```
#if 0
```

#*ifndef* ergibt *True*, falls *Bezeichner* nicht definiert ist, so daß die Zeile

```
#ifndef Bezeichner
```

denselben Effekt hat wie

```
#if 0
```

wenn *Bezeichner* gegenwärtig definiert ist, und denselben Effekt wie

```
#if 1
```

wenn *Bezeichner* gegenwärtig undefiniert ist.

Die Syntax entspricht im übrigen der bereits erläuterten von *#if*, *#elif*, *#else* und *#endif*.

Ein Bezeichner, der als NULL definiert wurde, wird als definiert betrachtet.

Die Direktive #line

Die *#line*-Direktive wird verwendet, um Zeilennummern in ein Programm für eine Kreuzreferenz oder zur Fehlerprüfung einzufügen. Wenn Ihr Programm aus Abschnitten besteht, die aus anderen Programmdateien stammen, dann ist es oft sinnvoll, diese Abschnitte mit den Zeilennummern der Originaldatei zu versehen statt mit den sequentiellen Zeilennummern des zusammengesetzten Programms. Die Syntax hierfür

#line *Integer_Konstante* <*"Dateiname"*>

kennzeichnet, daß die folgende Zeile des Programms von der Zeilennummer *Integer_Konstante* aus *Dateiname* stammt. Sobald *Dateiname* registriert ist, können nachfolgende *#line*-Direktiven, die sich auf diese Datei beziehen, auf die explizite Angabe des Arguments *Dateiname* verzichten, zum Beispiel:

```
// TEMP.C: Ein Beispiel für die #line-Direktive
#include <stdio.h>
#line 4 "junk.c"
void main()
{
    printf(" in Zeile %  d von % s",__LINE__,__FILE__);
#line 12 "temp.c"
    printf("\n");
    printf(" in Zeile %  d von % s",__LINE__,__FILE__);
#line 8
    printf("\n");
    printf(" in Zeile %  d von % s",__LINE__,__FILE__);
}
```

Die Einbindung von stdio.h hat zur Folge, daß die vom Präprozessor produzierte Datei recht groß sein wird.

Wenn Sie TEMP.C compilieren und starten, dann erhalten Sie

```
in Zeile 6 von junk.c
in Zeile 13 von temp.c
in Zeile 9 von temp.c
```

Makros werden in *#line*-Argumenten genauso wie in *#include*-Direktiven erweitert. Die *#line*-Direktive wird üblicherweise von Zusatzprogrammen, die C-Programmtext erzeugen, verwendet und weniger von C-Programmierern.

Die Direktive #error

Diese Direktive hat die Syntax

#error *Fehlermeldung*

und produziert die Meldung

```
Error:Dateiname Zeile: Error directive: Fehlermeldung
```

Diese Direktive wird gewöhnlich in einer Bedingungsdirektive eingebettet, die ein unerwünschtes Ereignis zum Zeitpunkt der Compilierung abfangen soll. Bei normalem Verlauf würde die Prüfung der Bedingung *False* ergeben. Wenn die Bedingung *True* ergibt, soll eine Fehlermeldung ausgegeben und die Compilierung augenblicklich unterbrochen werden.

Beispielsweise so: Mit #*define MEINWERT* definieren Sie ein Makro, das entweder den Wert 1 oder 0 annehmen kann. Sie können dann folgende Bedingung in den Quelltext aufnehmen, um zu testen, ob *MEINWERT* einen unerwünschten Wert annimmt:

```
#if (MEINWERT != 0 && MEINWERT != 1)
#error MEINWERT muß entweder den Wert 0 oder 1 haben
#endif
```

Die Direktive #pragma

Diese Direktive erlaubt die Definition implementationsspezifischer Präprozessor-Direktiven in der Form:

#pragma *Direktiven-Name*

Darüber kann Turbo C++ beliebige eigene Direktiven definieren, ohne dabei mit anderen Compilern in Konflikt zu kommen, die #*pragma* ebenfalls unterstützen. Wenn ein Compiler *Direktiven-Name* nicht kennt, dann ignoriert er diese Direktive, ohne Warnungen und Fehlermeldungen auszugeben.

Turbo C++ unterstützt die folgenden #*pragma*-Direktiven:

- #pragma argsused
- #pragma exit
- #pragma hdrfile
- #pragma hdrstop
- #pragma saveregs
- #pragma startup

#pragma argsused

Dieses Pragma ist nur zwischen Funktionsdefinitionen erlaubt und beeinflußt nur die nächste Funktion. Dieses Pragma unterbindet die Warnung:

```
"Parameter Name is never used in function Funktionsname"
```

#pragma exit und #pragma startup

Mit diesen Pragmas kann man festlegen, daß eine oder mehrere Funktion(en) beim Programmstart (also vor *main*) oder beim Beenden des Programms (also unmittelbar vor der Funktion *_exit*) aufgerufen werden.

Die Pragmas haben folgende Syntax:

#pragma startup *Funktionsname <Priorität>*
#pragma exit *Funktionsname <Priorität>*

Die angegebene Funktion muß *vorher* deklariert worden sein, darf keine Argumente übernehmen und muß **void** liefern. Sie wird also so deklariert:

```
void funk(void);
```

Der optionale Parameter *Priorität* sollte ein Integerwert zwischen 64 und 255 sein. 0 hat die höchste Priorität. Die Werte 0 bis 63 werden von den C-Bibliotheken benutzt und sollten nicht vom Benutzer verwendet werden. Funktionen mit höheren Prioritäten werden beim Programmstart zuerst und beim Programmende zuletzt aufgerufen.

Wenn Sie keine Priorität festlegen, erhält die bezeichnete Funktion automatisch die Priorität 100. Betrachten Sie folgendes Beispiel:

```
#include <stdio.h>

void startFunk(void)
{
   printf("So geht's los.\n");
}

#pragma startup startFunk 64
// Priorität 64 --> wird beim Start als erste aufgerufen
void endeFunk(void)

{
printf("So hört's auf.\n");
}

#pragma exit endeFunk
// Voreingestellte Priorität 100
void main (void)
{
#if defined(_Windows)
```

```
    _InitWinCrt();
#endif

    printf("So ist's mitten drin.\n");
}
```

Beachten Sie, daß die Funktionen, auf die in *#pragma startup* und *#pragma exit* Bezug genommen wird, vorher deklariert worden sein müssen.

#pragma hdrfile

Diese Direktive bestimmt den Namen der Datei, in der die vorcompilierten Header gespeichert werden. Der voreingestellte Name für diese Datei lautet TCDEF.SYM. Die Direktive hat folgende Syntax:

```
#pragma hdrfile "Dateiname.SYM"
```

Wenn Sie keine vorcompilierten Header verwenden, hat diese Direktive keine Auswirkung. Mit der Option *Vorcompilierte Header* im Menü *Optionen/Compiler/Code-Generierung* können Sie den Namen der Datei ändern, die die vorcompilierten Header aufnehmen soll. In Anhang D »Vorcompilierte Header-Dateien« im *Turbo C++für Windows Benutzerhandbuch* finden Sie weitere Informationen zu diesem Punkt.

#pragma hdrstop

Diese Direktive schließt die Liste der Header-Dateien, die für die Vorcompilierung vorgesehen sind. Mit Hilfe von *#pragma hdrstop* kann der für vorcompilierte Header erforderliche Festplattenplatz reduziert werden. (In Anhang D »Vorcompilierte Header-Dateien« im *Benutzerhandbuch* finden Sie eine Erläuterung dieser Dateien.)

#pragma saveregs

Dieses Pragma garantiert, daß eine Funktion im Modell *Huge* die Segmentregister des Prozessors unverändert läßt. Diese Direktive wird manchmal für Aufrufe von Maschinenprogrammen gebraucht. Sie sollte unmittelbar vor der Funktionsdefinition stehen und gilt nur für die unmittelbar folgende Funktion.

Vordefinierte Makros

Turbo C++ definiert eine Reihe von globalen Bezeichnern (die im folgenden erläutert werden) vor. Außer __cplusplus und _WINDOWS beginnt und endet jeder dieser Bezeichner mit zwei Unterstrichen (__). Diese Makros werden auch als *manifeste Konstanten* bezeichnet.

__CDECL__

Dieses Makro gehört speziell zur Familie der Borland C- und C++ Compiler. Es signalisiert, daß das Schaltfeld *C* im Dialogfenster *Generierung von Entry-|Exit-code* nicht verwendet wurde. Es liefert die Konstante 1, wenn mit ausgeschalteter Option wird, ansonsten ist das Ergebnis undefiniert.

Die folgenden vier Symbole stehen für das bei der Compilierung verwendete Speichermodell:

```
__COMPACT__          __MEDIUM__
__LARGE__             __SMALL__
```

Nur jeweils eine dieser Konstanten ist bei einer Compilierung definiert, die anderen Modelle sind per Definition undefiniert. Wenn Sie z.B. mit Speichermodell *Small* compilieren, ist nur das Makro *__SMALL__* definiert, die anderen bleiben undefiniert, so daß die Direktive:

```
#if defined(__SMALL__)
```

den Wert *True* ergibt, während

```
if defined(__LARGE__)
```

(und alle anderen Makros) *False* ergeben. Der aktuelle Wert des jeweils definierten Makros ist 1.

__cplusplus

In C++ ist das Makro als 1 definiert, in allen anderen Fällen ist es undefiniert. Es erlaubt Ihnen ein Modul zu schreiben, das einmal als C und einmal als C++ übersetzt werden kann. Mit bedingter Compilierung können Sie steuern, welche C- und C++ Programmteile betroffen sein sollen.

__DATE__

Das Makro liefert das Datum, zu dem der Präprozessor die Bearbeitung der aktuellen Datei begonnen hat, zurück (als Stringkonstante).

Jede Expansion von *__DATE__* innerhalb einer Datei liefert denselben Wert zurück – unabhängig davon, wie lange die Bearbeitung durch den Präprozessor dauert. Das Datum wird im Format *mmm tt jjjj* ausgegeben, wobei *mmm* für ein

dreibuchstabiges Kürzel der Monatsnamen steht (Jan, Feb, Mar etc.), *tt* für zwei Dezimalziffern (1 bis 31) bzw. ein Leerzeichen und eine folgende Ziffer innerhalb der ersten neun Tage jedes Monats, und *jjjj* für das Jahr (1990, 1991...).

__DLL__

Dieses Makro gehört speziell zur Familie der Borland C- und C++Compiler. Es ist als 1 definiert, wenn Sie ein Modul über das Schaltfeld *Windows DLL: alle Funktionen exportierbar* (*Optionen/Compiler/Entry-/Exitcode*) in der IDE compilieren, um Code für dynamische Windows-Linkbibliotheken zu erzeugen; in allen anderen Fällen bleibt das Makro undefiniert.

__FILE__

Das Makro liefert den Namen der aktuellen Quelltextdatei als Stringkonstante zurück. Der zurückgelieferte String ändert sich während der Bearbeitung einer #*include*-Anweisung automatisch bzw. kann über #*line* explizit verändert werden.

__LINE__

Das Makro liefert die Nummer der aktuellen Quelltextzeile, die gerade bearbeitet wird (als dezimale Konstante). Die erste Zeile eines Quelltextes hat normalerweise die Nummer 1.

Die Direktive #*line* kann dies beeinflussen. (Vgl. dazu den Abschnitt »Die Direktive #line« weiter vorne in diesem Kapitel.)

__MSDOS__

Dieses Makro gehört speziell zur Familie der Borland C- und C++Compiler und liefert grundsätzlich die Integerkonstante 1 zurück.

__PASCAL__

Dieses Makro gehört speziell zur Familie der Borland C- und C++Compiler. Es signalisiert, daß die Pascal-Aufrufkonvention (*Optionen/Compiler/Entry-|Exitcode*) verwendet wurde und liefert die Konstante 1, wenn diese Konvention benutzt wird, ansonsten ist das Makro undefiniert.

__STDC__

Das Makro liefert die Konstante 1, wenn Sie mit dem *ANSI*-Schalter im Menü *Quellcode* compilieren. Ansonsten ist dieses Makro undefiniert.

__TCPLUSPLUS__

Dieses Makro gehört speziell zur Familie der Borland C- und C++Compiler. Es ist nur für C++Compilierung als hexadizimale Konstante 0x0300 definiert. Dieser numerische Wert wird in späteren Versionen erhöht.

__TEMPLATES__

Dieses Makro gehört speziell zur Familie der Borland C- und C++Compiler. Für C++ Dateien ist es mit 1 definiert (was besagt, daß Turbo C++ Templates unterstützt); ansonsten ist das Makro undefiniert.

__TIME__

Das Makro liefert die Uhrzeit zurück, zu der der Präprozessor die Bearbeitung der aktuellen Datei begonnen hat (als Stringkonstante).

Wie bei *__DATE__* liefert jede Expansion von *__TIME__* innerhalb einer Datei denselben Wert zurück – unabhängig davon, wie lange die Bearbeitung durch den Präprozessor dauert. Das Format der zurückgelieferten Stringkonstanten ist *hh:mm:ss*, wobei *hh* für die Stunden (00 bis 23) steht, *mm* für die Minuten (00 bis 59) und *ss* für die Sekunden (00 bis 59).

__TURBOC__

Dieses Makro gehört speziell zur Familie der Borland C- und C++Compiler. Es ist als hexadizimale Konstante 0x0400 definiert. Dieser numerische Wert wird in späteren Versionen erhöht.

_Windows

Dieses Makro zeigt an, daß Windows-spezifischer Programmtext erzeugt werden soll. Dieses Makro ist per Voreinstellung für Turbo C++definiert.

5. C++ *Streams*

Dieses Kapitel ist in zwei Teile gegliedert: eine kurze Übersicht über den Gebrauch der C++Stream-I/O und einen Referenzteil zu ausgewählten Klassen aus der C++Stream-Klassenbibliothek.

Stream I/O (Eingabe/Ausgabe) in C++ (im allgemeinen spricht man von *streams* oder *iostreams*) stellt den gesamten Funktionsumfang der *stdio*-Bibliothek in C zur Verfügung. *iostreams* werden verwendet, um typisierte Objekte in lesbaren Text zu konvertieren und umgekehrt. Ebenso können Streams binäre Daten lesen und schreiben. Die Sprache C++ erlaubt es Ihnen, Ein-/Ausgabefunktionen und Operatoren zu definieren, die dann automatisch für entsprechende benutzerdefinierte Typen aufgerufen werden.

Was ist ein Stream?

Ein *Stream* ist definiert als der abstrakte Datenfluß von einer *Quelle* (oder *Produzenten*) zu einem *Ziel* (oder *Konsumenten*). Synonym verwenden wir zur Beschreibung des Einlesens von Zeichen aus einer Quelle auch die Verben *übernehmen*, *eingeben* oder *holen* und zur Beschreibung der Übergabe an das Zielmedium *einfügen*, *schreiben* oder *speichern*.

Es stehen Klassen zur Verfügung, die Puffer im Arbeitsspeicher (iostream.h), Dateien (fstream.h) und Strings (strstrea.h) als Quellen oder Ziele (oder beides) unterstützen.

Die I/O-Stream-Bibliothek

Die I/O-Stream-Bibliothek (definiert in iostream.h) besteht aus zwei parallelen Klassenfamilien: Eine ist abgeleitet von *streambuf*, die andere von *ios*. Beide sind low-level-Klassen, erfüllen jedoch unterschiedliche Aufgaben. Alle Stream-Klassen stammen von einer dieser beiden Klassen ab. Der Zugriff von *ios*-Klassen auf *streambuf*-Klassen erfolgt über einen Zeiger.

Die Klasse streambuf

Die *streambuf*-Klasse stellt eine Schnittstelle zu Peripheriegeräten zur Verfügung. Sie bietet Methoden zum Puffern und Bearbeiten von Streams, wenn wenig oder keine Formatierung erforderlich ist. *streambuf* ist eine nützliche Basisklasse, die von anderen Teilen der iostream-Bibliothek benutzt wird. Sie können auch Klassen für eigene Funktionen und Bibliotheken aus diesen Klassen ableiten. Die Klassen *filebuf* und *strstreambuf* sind von *streambuf* abgeleitet.

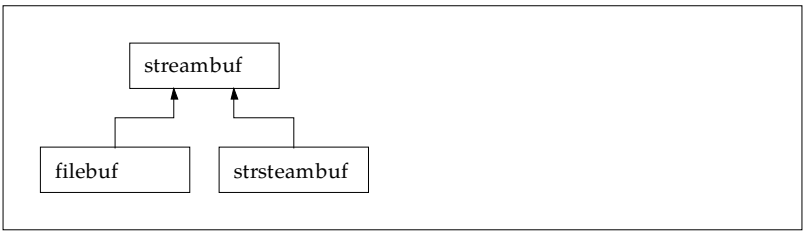


Abbildung 5.1 Die Klasse streambuf und die von ihr abgeleiteten Klassen

Die Klasse ios

Die Klasse *ios* (und daher auch alle von ihr abgeleiteten Klassen) enthält einen Zeiger auf ein *streambuf*-Objekt. Darauf aufbauend führt sie formatierte Eingaben und Ausgaben und eine Fehlerprüfung durch.

In Abbildung 5.2 finden Sie ein Ableitungsdiagramm für die gesamte Klassenfamilie von *ios*. So ist z.B. die Klasse *ifstream* von den Klassen *istream* und *fstreambase* abgeleitet, *istrstream* ist eine Ableitung von *istream* und *strstreambase*. Weil hier von Mehrfachvererbung reichlich Gebrauch gemacht wird, ist das Diagramm nicht ganz einfach zu lesen *Mehrfachvererbung* bedeutet, daß eine Klasse mehrere Vorfahren haben kann. (C++ bedient sich zur Vermeidung von Mehrfach-Deklarationen der *virtuellen Vererbung*.) Zum Beispiel sind alle Elemente (Daten und Funktionen) von *iostream*, *istream*, *ostream*, *fstreambase* und *ios* Bestandteile von *fstream*. Alle Klassen in der *ios*-Vernetzung verwenden eine *streambuf*-Klasse (oder eine *filebuf*- oder *strstreambuf*-Klasse, die von *streambuf* abstammen) als Quelle und/oder Ziel.

Beim Start eines C++ Programms sind die folgenden vier Streams, die als Objekte der *withassign*-Klassen deklariert sind, geöffnet:

```
extern istream_withassign cin;   // Entspricht stdin
extern ostream_withassign cout;  // Entspricht stdout
extern ostream_withassign cerr;  // Entspricht stderr
extern ostream_withassign clog;  // Gepufferte Variante von cerr
```

Hinweis: Der allgemein üblichen Praxis folgend zeigen die Pfeile in dieser Abbildung jeweils vom Nachfahren zum Vorfahren.

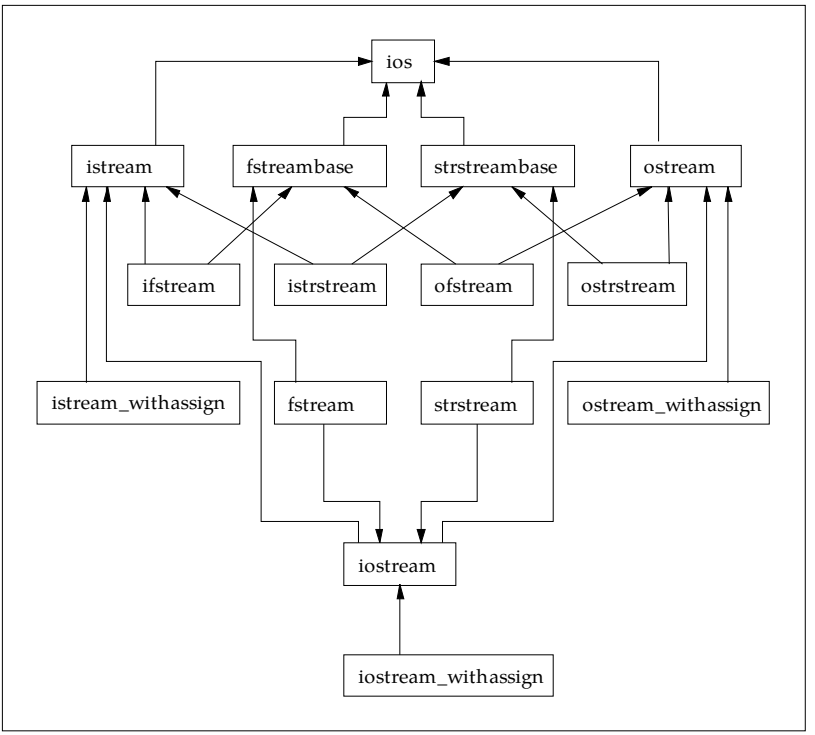


Abbildung 5.2 Die Klasse ios und ihre Nachfahren

Ausgabe

Die Stream-Ausgabe wird mit dem *Übergabe*-Operator << erzeugt. Der standardmäßige Linksschieben-Operator << ist für Ausgabe-Operationen überladen. Sein linker Operand ist ein Objekt der Klasse *ostream*, sein rechter Operand ist ein beliebiger Typ, für den Stream-Ausgabe definiert wurde (d.h. vordefinierte Typen oder Typen, für die Sie den Übergabe-Operator überladen haben).

Zum Beispiel übergibt `cout << "Hallo!\n";` die Zeichenkette *"Hallo!"* an *cout* (den Standardausgabe-Stream, normalerweise den Bildschirm), gefolgt von einer neuen Zeile.

Der Operator << ist links-assoziativ und liefert eine Referenz auf das *ostream*-Objekt, für das er aufgerufen wurde. Dies erlaubt es, mehrere Übergaben aneinanderzureihen:

```
int i = 8;
double d = 2.34;
    cout << "i=" << i << ", d= " << d << "\n";
```

Dies wird `i = 8, d = 2.34` zur Standardausgabeeinheit schreiben.

Vordefinierte Typen

Folgende Datentypen werden direkt unterstützt: **char**, **short**, **int**, **long**, **char*** (wird als String behandelt), **float**, **double**, **long double** und **void***. Integertypen werden gemäß den voreingestellten Regeln für *printf* umgewandelt (es sei denn, Sie haben diese Regeln durch das Setzen verschiedener *ios*-Flags geändert).

Z.B. ergeben die beiden folgenden Ausgabeanweisungen dasselbe Ergebnis:

```
int i;
long l;
cout << i << " " << 1;
printf("%d %ld", i, l);
```

Der Zeiger-Übergabeoperator (void*) wird zur Anzeige von Zeigern verwendet:

```
int i;
cout << &i;     // Zeiger in hexadezimaler Form anzeigen
```

Hinweis: Im Abschnitt »Stream-Klassenreferenz« weiter hinten in diesem Kapitel finden Sie eine Beschreibung der Klasse *ostream* für weitere Ausgabefunktionen.

Formatierung der Ausgabe

Die Formatierung von Eingaben und Ausgaben wird durch verschiedene *Formatstatus*-Flags bestimmt, die in der Klasse *ios* definiert sind. Es stehen folgende Formatstatus-Flags zur Verfügung:

```
public:
enum {
  skipws,       // ignoriert Whitespace bei der Eingabe
  left   ,      // linksbündige Ausgabe
  right,        // rechtsbündige Ausgabe
  internal,     // Füllzeichen nach Vorzeichen- oder
                // Basisindikator
  dec,          // Dezimalkonvertierung
  oct,          // Oktalkonvertierung
  hex,          // hexadezimale Konvertierung
  showbase,     // zeigt Basisindikator bei der Ausgabe
  showpoint,    // zeigt Dezimalpunkt (fp-Ausgabe)
  uppercase,    // Hex-Ausgabe mit Großbuchstaben
  showpos,      // zeigt '+' bei positiven Integerzahlen
  scientific,   // benutzt das 1.2345E2 fp-Format
  fixed,        // benutzt das 123.45 fp-Format
  unitbuf,      // leert alle Streams nach der übergabe
  stdio,        // leert stdout und stderr nach der übergabe
};
```

Diese Flags werden mit den Elementfunktionen *flags, setf* und *unsetf* gelesen und gesetzt.

Hinweis: Eine Erläuterung dieser Funktionen finden Sie im Abschnitt »Stream-Klassenreferenz« bei der Klasse *ios* weiter hinten in diesem Kapitel.

Manipulatoren

Ein einfacherer Weg, Formatvariabeln zu ändern, ist die Benutzung eines speziellen, funktionsähnlichen Operators, genannt *Manipulator*. Manipulatoren nehmen eine Stream-Referenz als Argument entgegen und liefern eine Referenz an denselben Stream zurück. Das bedeutet, daß Manipulatoren in eine Kette von Übergaben (oder Übernahmen) eingebettet werden können.

Damit kann – als Seiteneffekt – der Stream-Status geändert werden, ohne daß tatsächlich irgendwelche Übergaben (oder Übernahmen) stattfinden. Zum Beispiel produziert

```
#include <iostream.h>
#include <iomanip.h>     // Für parametrisierte Manipulatoren
                         // erforderlich

int main(void) {
int i = 6789, j = 1234, k = 10;

   cout << setw(6) << i << j << i << k << j;
   cout << "\n";
   cout << setw(6) << i << setw(6) << j << setw(6) << k;
   return(0);
   }
```

folgende Ausgabe:

```
678912346789101234
6789  1234    10
```

setw ist ein in iomanip.h deklarierter *parametrisierter Manipulator*. Andere parametrisierte Manipulatoren – *setbase, setfill, setprecision, setiosflags* sowie *resetiosflags* – arbeiten in gleicher Weise. Um diese benutzen zu können, muß Ihr Programm iomanip.h einbinden. Sie können auch eigene Manipulatoren ohne Parameter schreiben:

```
#include <iostream.h>

// Tabuliert und versieht die Ausgabe mit einem Dollarzeichen.
ostream& money( ostream& output ) {
   return output << "\t$";
   }

int main(void) {
   float owed = 1.35, earned = 23.1;
   cout << money << owed << money << earned;
   return(0);
   }
```

Obiges Programm ergibt folgende Ausgabe:

```
$1.35   $23.1
```

Die *nicht-parametrisierten* Manipulatoren *dec*, *hex* und *oct* (die in iostream.h deklariert sind) nehmen keine Argumente entgegen und ändern (dauerhaft) die Konvertierungsbasis:

```
int i = 36
cout  << dec << i << " " << hex << i << " " << oct << i << endl;
cout << dec; // auf dezimale Ausgabe zurücksetzen
// zeigt 36 24 44
```

Tabelle 5.1 Manipulatoren

Manipulator	Aktion
dec	Setzt das Flag für Dezimalkonvertierung.
hex	Setzt das Flag für Hexadezimalkonvertierung .
oct	Setzt das Flag für Oktalkonvertierung.
ws	Entfernt Whitespace-Zeichen.
endl	Fügt Neuezeile ein und leert den Stream-Puffer.
ends	Fügt eine abschließende Null in Strings ein.
flush	Leert einen *ostream*.
setbase(int *n*)	Setzt das Konvertierungsformat auf Basis *n* (0, 8, 10 oder 16). 0 ist der Vorgabewert: dezimale Ausgabe, C-Regeln für Integer bei der Eingabe.
resetiosflags(*long f*)	Löscht die von *f* bestimmten Format-Bits.
setiosflags(*long f*)	Setzt die von *f* bestimmten Format-Bits.
setfill(*int c*)	Setzt das Füllzeichen auf *c*.
setprecision(*int c*)	Setzt die Gleitkommagenauigkeit auf *c* Stellen.
setw(*int n*)	Setzt die Feldbreite auf *n*.

Der Manipulator *endl* fügt ein Neuezeile-Zeichen ein und leert den Stream. Sie können einen *ostream* auch jederzeit so leeren:

```
ostream << flush;
```

Füllen der Feldbreite

Das Füllzeichen und die Richtung des Auffüllens hängen vom Setzen der Füllzeichen und den Flags *left*, *right* und *internal* ab. Das vorgegebene Füllzeichen ist ein Leerzeichen. Es kann mit der Funktion *fill* geändert werden:

```
int i = 123;
cout.fill('*');
cout.width(6);
cout << i;        // zeigt ***123
```

Gemäß der Voreinstellung wird die Ausgabe rechtsbündig justiert, Füllzeichen werden also links eingefügt. Sie können diese Voreinstellungen (und andere Format-Flags) mit den Funktionen *setf* und *unsetf* ändern:

```
int i = 56;
...
cout.width(6);
cout.fill('#');
cout.setf(ios::left,ios::adjustfield);
cout << i;        // zeigt 56####
```

Das zweite Argument, *ios::adjustfield*, bestimmt, welche Bits zu setzen sind und das erste Argument, *ios::left*, wie sie gesetzt werden sollen. Alternativ dazu können Sie die Manipulatoren *setfill*, *setiosflags* und *resetiosflags* benutzen, um das Füllzeichen und den Füll-Modus zu ändern. Eine Erläuterung der *ios*-Datenelemente finden Sie weiter hinten bei der Beschreibung von *ios*.

Eingabe

Stream-Eingabe funktioniert ähnlich wie die Ausgabe, benutzt aber den überladenen Rechtsschieben-Operator >>, der auch *Übernahme*-Operator genannt wird. Der linke Operand von >> ist ein Objekt der Klasse *istream*. Wie bei der Ausgabe kann der rechte Operand von irgendeinem Typ sein, für den Stream-Eingabe definiert wurde.

Per Voreinstellung ignoriert >> Whitespace-Zeichen (welche das sind, legt die in der Datei ctype.h definierte *ispace*-Funktion fest) und liest Zeichen, die zum Typ des Eingabe-Objekts passen. Das Ignorieren von Whitespace-Zeichen wird durch das Flag *ios::skipws* in der Formatstatus-Auflistung gesteuert. Das *skipws*-Flag ist normalerweise gesetzt, um Whitespaces zu ignorieren. Das Löschen dieses Flags (zum Beispiel mit *setf*) schaltet diese Eigenschaft aus. Beachten Sie auch den speziellen »Ziel«-Manipulator *ws*, mit dem Sie Whitespaces entfernen können. Betrachten Sie das folgende Beispiel:

```
int i;
double d;
cin >> i >> d;
```

Bei Ausführung der letzten Zeile überliest das Programm zunächst alle führenden Whitespace-Zeichen. Danach wird der Integerwert (*i*) gelesen. Jedes auf den Integerwert folgende Whitespace wird ignoriert. Schließlich wird der Gleitkomma-Wert (*d*) gelesen.

Für den Typ **char** (**signed** oder **unsigned**) hat der Operator >> folgenden Effekt: Überlesen von Whitespaces und Speichern des nächsten Nicht-Whitespace-Zeichens. Wenn Sie das nächste Zeichen haben wollen, gleichgültig, ob es ein Whitespace ist oder nicht, so verwenden Sie eine der *get*-Elementfunktionen (vgl. dazu die Erläuterung der *istream*-Klasse im Abschnitt »Stream-Klassenreferenz« weiter hinten in diesem Kapitel).

Für den Typ **char*** (der als String behandelt wird) hat der Operator >>

folgenden Effekt: Whitespace wird überlesen und alle folgenden Nicht-Whitespace-Zeichen werden gespeichert, bis ein weiteres Whitespace-Zeichen gefunden wird. Danach wird ein abschließendes Null-Zeichen angefügt. Achten Sie darauf, daß kein »Überlauf« entsteht.

Die vorgegebene Feldbreite von Null (das bedeutet unbeschränkte Breite) kann geändert werden, wenn *width* wie folgt benutzt wird:

```
char array[SIZE];
cin.width(sizeof(array));
cin >> array // vermeidet überlauf
```

Bei vordefinierten Typen wird bei der Eingabe im Zielpuffer nichts gespeichert und der *istream*-Status wird auf Fehler gesetzt, wenn das Eingabeende erreicht wurde, bevor ein Nicht-Whitespace-Zeichen gefunden wurde. Wenn also das Ziel vorher nicht initialisiert war, wird dies auch jetzt nicht der Fall sein.

Ein-/Ausgabe von benutzerdefinierten Typen

Wenn Sie selbstdefinierte Typen speichern oder lesen wollen, müssen Sie die entsprechenden Übergabe- und Übernahme-Operatoren überladen. Sehen Sie sich dazu folgendes Beispiel an:

```
#include <iostream.h>

struct info {
   char *name;
   double val;
   char *units;
   };

// Sie können << für die Ausgabe folgendermaßen überladen:
ostream& operator << (ostream& s, info& m) {
   s << m.name << " " << m.val << " " << m.units;
   return s;
   };

// Sie können >> für die Eingabe folgendermaßen überladen:
istream& operator >> (istream& s, info& m) {
   s >> m.name >> m.val >> m.units;
   return s;
   };

int main(void) {
   info x;
   x.name = new char[15];
   x.units = new char[10];

   cout << "\nInput name, value and units:";
   cin >> x;
```

```
cout << "\nMy input:" << x;
return(0);
}
```

Einfache Datei-Ein-/Ausgabe

Die Klasse *ofstream* erbt die Übergabe-Operationen von *ostream*, während *ifstream* die Übernahme-Operationen von *istream* erbt. Die beiden Klassen enthalten ferner Konstruktoren und Elementfunktionen für die Erzeugung von Dateien und die Handhabung von Datei-Ein-/Ausgabe. Sie müssen fstream.h in alle Programme, die diese Klassen benutzen, einbinden. Betrachten Sie das folgende Beispiel, das die Datei FILE.IN in die Datei FILE.OUT kopiert:

```
#include <fstream.h>

int main(void)  {
   char ch;
   ifstream f1("FILE.IN");
   ofstream f2("FILE.OUT");

   if (!f1) cerr << "Cannot open FILE.IN for input";
   if (!f2) cerr << "Cannot open FILE.OUT for output";
   while (f2 && f1.get(ch))
      f2.put(ch);
   return(0);
   }
```

Beachten Sie, daß der entsprechende Stream-Fehlerstatus gesetzt wird, wenn der *ifstream*- oder *ofstream*-Konstruktor die spezifizierte Datei nicht öffnen kann.

Die Konstruktoren erlauben es Ihnen, einen Datei-Stream zu deklarieren, ohne eine bestimmte Datei zu spezifizieren. Später können Sie dann den Datei-Stream einer ganz bestimmten Datei zuordnen:

```
ofstream ofile;          // erzeugt einen Ausgabe-Stream
...
ofile.open("payroll");   // verbindet ofile-Stream mit
                         // der Datei "payroll"
...
ofile.close();           // schließt den ofile-Stream
ofile.open("employee");  // ofile kann wieder benutzt werden
```

Dateien werden per Voreinstellung im Text-Modus geöffnet. Dies bedeutet, daß bei der Eingabe CR/LF-Sequenzen in das '\n'-Zeichen umgewandelt werden. Bei der Ausgabe wird das '\n'-Zeichen in eine CR/LF-Sequenz umgewandelt. Diese Übersetzung erfolgt nicht im binären Modus. Der Dateimodus wird mit einem zweiten optionalen Parameter der Funktion *open* gesetzt, der aus der folgenden Tabelle ausgewählt werden kann:

Tabelle 5.2 Dateimodi

Modus-Bit	Aktion
ios::app	Hängt Daten an – schreibt immer an das Dateiende.
ios::ate	Positioniert auf das Dateiende nach *open*.
ios::in	Öffnet für Eingabe (implizit für *ifstreams*).
ios::out	Öffnet für Ausgabe (implizit für *ofstreams*).
ios::binary	Öffnet die Datei im binären Modus.
ios::trunc	Verwirft den Inhalt der Datei, wenn sie existiert (implizit, wenn *ios::out* angegeben ist und weder *ios::ate* noch *ios::app* angegeben sind).
ios::nocreate	Fehler bei *open*, wenn die Datei nicht existiert.
ios::noreplace	Fehler bei *open* für die Ausgabe, wenn die Datei existiert, es sei denn, *ate* oder *app* ist gesetzt.

String-Streams

Die in strstrea.h definierten Funktionen unterstützen eine Formatierung im Speicher, ähnlich wie *sscanf* und *sprintf*, sind aber weitaus flexibler. Alle *istream*-Funktionen stehen der Klasse *istrstream* (*i*nput *str*ing *stream*) zur Verfügung. Ähnlich ist es bei der Ausgabe: *ostrstream* wird von *ostream* abgeleitet. Das untenstehende Beispielprogramm liest eine Textdatei mit folgendem Format:

```
101 191 Cedar Chest
102 1999.99 Livingroom Set
```

Jede Zeile wird in drei Komponenten zerlegt: eine Integer-ID, einen Gleitkomma-Preis und eine Beschreibung. Folgende Ausgabe wird erstellt:

```
1: 101 191.00   Cedar Chest
2: 102 1999.99  Livingroom Set
```

Das Programm sieht folgendermaßen aus:

```
#include <fstream.h>
#include <strstrea.h>
#include <iomanip.h>
#include <string.h>

void main(int argc, char **argv) {
   int id;
   float amount;
   char description[41];
   ifstream inf(argv[1]);

   if (inf) {
      char inbuf[81];
      int lineno = 0;
```

```
        // Festen Dezimalpunkt einstellen
        cout.setf(ios::fixed, ios::floatfield);

      // Gleitkommazahlen sollen mit Dezimalpunkt ausgegeben werden
      cout.setf (ios::showpoint);

      while (inf.getline (inbuf,81)) {
        // 'ins' ist der String-Stream:
        istrstream ins (inbuf,strlen (inbuf));
        ins >> id >> amount >> ws;
        ins.getline (description,41);   // Behält die neue Zeile!
        cout << ++lineno << ": "
           << id << '\t'
           << setprecision (2) << amount << '\t'
           << description << "\n";
      }
    }
    return(0);
}
```

Beachten Sie die Verwendung der Format-Flags und Manipulatoren im obigen Beispiel. Der Aufruf von *setf* und der Manipulator *setprecision* ermöglichen es, Gleitkommazahlen im Währungsformat auszudrucken. Der Manipulator *ws* ignoriert Whitespace-Zeichen, bevor der Beschreibungs-String eingelesen wird.

Stream-Klassenreferenz

Die Stream-Klassenbibliothek von C++umfaßt weit mehr Klassen, als wir hier aufführen können. Wir haben uns auf die wesentlichen Klassen beschränkt; wie üblich sind diese alphabetisch angeordnet.

Die folgende Aufstellung zeigt, welche Klassen zu welcher Header-Datei gehören:

constrea.h	*conbuf, constream*
iostream.h	*ios, iostream, iostream_withassign, istream, istream_withassign, ostream, ostream_withassign, streambuf*
fstream.h	*filebuf, fstream, fstreambase, ifstream, ofstream.*
strstrea.h	*istrstream, ostrstream, strstream, strstreambase, strstreambuf.*

filebuf <fstream.h>

Spezialisiert *streambuf* zur Behandlung von Dateien.

Konstruktor `filebuf();`

Erzeugt ein *filebuf*-Objekt, das mit keiner Datei verbunden ist.

Konstruktor `filebuf (int fd);`

Erzeugt ein *filebuf*-Objekt, das über den Datei-Deskriptor *fd* mit einer Datei verbunden ist.

Konstruktor `filebuf (int fd, char*, int n);`

Erzeugt ein *filebuf*-Objekt, das mit einer Datei verbunden ist und den angegebenen Zeichenpuffer der Länge *n* verwendet.

Element-funktionen

attach `filebuf* attach (int);`

Verbindet das *filebuf*-Objekt mit einer geöffneten Datei über deren Deskriptor.

close `filebuf* close();`

Leert den Puffer und schließt die Datei. Liefert im Fehlerfall 0 zurück.

fd `int fd ();`

Liefert den Datei-Deskriptor oder EOF zurück.

is_open `int is_open();`

Liefert einen Wert ungleich Null, wenn die Datei geöffnet ist.

open `filebuf* open(const char*, int mode, int prot= filebuf::openprot);`

Öffnet die gegebene Datei und verbindet sie mit dem Objekt.

overflow `virtual int overflow(int = EOF);`

Leert den Puffer in sein Ziel. Jede abgeleitete Klasse sollte die erforderlichen Aktionen definieren.

seekoff `virtual streampos seekoff(streamoff, ios::seek_dir, int);`

Verschiebt den Dateizeiger relativ zur aktuellen Position.

setbuf `virtual streambuf* setbuf(char*, int);`

Bestimmt einen Puffer für das aufrufende *filebuf*-Objekt.

sync `virtual int sync();`

Gewährleistet die Konsistenz zwischen internen Datenstrukturen und der externen Stream-Repräsentation.

underflow `virtual int underflow();`

Macht die Eingabe verfügbar. Die Funktion wird aufgerufen, wenn keine weiteren Daten im Eingabepuffer vorhanden sind.

Jede abgeleitete Klasse sollte die erforderlichen Aktionen definieren.

fstream <fstream.h>

Diese von *fstreambase* und *iostream* abgeleitete Klasse sorgt für gleichzeitige Ein- und Ausgabe in einer *filebuf*-Klasse.

Konstruktor `fstream();`

Erzeugt ein *fstream*-Objekt, das mit keiner Datei verbunden ist.

Konstruktor `fstream(const char*, int, int = filebuf::openprot);`

Erzeugt ein *fstream*-Objekt, öffnet eine Datei und verbindet sie mit *fstream*.

Konstruktor `fstream(int);`

Erzeugt ein *fstream*-Objekt und verbindet es mit einem Deskriptor einer geöffneten Datei.

Konstruktor `fstream(int, char*, int);`

Erzeugt ein *fstream*-Objekt, das mit einer geöffneten Datei verbunden ist, und verwendet den angegebenen Puffer.

Elementfunktionen

open `void open(const char*, int, int = filebuf::openprot);`

Öffnet eine Datei für das *fstream*-Objekt.

rdbuf `filebuf* rdbuf();`

Liefert den verwendeten Dateipuffer zurück.

fstreambase <fstream.h>

Diese von *ios* abgeleitete Klasse stellt Operationen für Datei-Streams zur Verfügung und dient als Basis für *fstream*, *ifstream* und *ofstream*.

Konstruktor `fstreambase();`

Erzeugt ein *fstreambase*-Objekt, das mit keiner Datei verbunden ist.

Konstruktor `fstreambase(const char*, int, int = filebuf::openprot);`

Erzeugt ein *fstreambase*-Objekt, öffnet eine Datei und verbindet sie mit *fstreambase*.

Konstruktor `fstreambase(int);`

Erzeugt ein *fstreambase*-Objekt und verbindet es mit einem Deskriptor einer geöffneten Datei.

Konstruktor `fstreambase(int, char*, int);`

Erzeugt ein *fstreambase*-Objekt, das mit einer geöffneten Datei verbunden ist, und verwendet den angegebenen Puffer.

Element-funktionen

attach `void attach(int);`

Stellt mit Hilfe des Deskriptors eine Verbindung zu einer geöffneten Datei her.

close `void close();`

Schließt die zugehörige Datei.

open `void open(const char*, int, int = filebuf::openprot);`

Öffnet eine Datei für ein *fstreambase*-Objekt.

rdbuf `filebuf* rdbuf();`

Liefert den verwendeten Puffer zurück.

setbuf `void setbuf(char*, int);`

Verwendet den angegebenen Puffer.

ifstream <fstream.h>

Diese von *fstreambase* und *istream* abgeleitete Klasse stellt Eingabeoperationen für eine *filebuf*-Klasse zur Verfügung.

Konstruktor `ifstream();`

Erzeugt ein *ifstream*-Objekt, das mit keiner Datei verbunden ist.

Konstruktor
```
ifstream(const char*, int = ios::in,
int = filebuf::openprot);
```

Erzeugt ein *ifstream*-Objekt, öffnet eine Eingabedatei im Protected Mode und verbindet sie mit *ifstream*. Der Inhalt der vorhandenen Datei bleibt erhalten; neue Einträge werden angefügt.

Konstruktor `ifstream(int);`

Erzeugt ein *ifstream*-Objekt und verbindet sie mit einer geöffneten Datei über deren Deskriptor.

Konstruktor `ifstream(int fd, char *, int);`

Erzeugt ein *ifstream*-Objekt, das mit einer geöffneten Datei verbunden ist, und verwendet den angegebenen Puffer.

Element-funktionen

open `void open(const char*, int, int = filebuf::openprot);`

Öffnet eine Datei für das *ifstream*-Objekt.

rdbuf `filebuf* rdbuf();`

Liefert den verwendeten Puffer zurück.

ios <iostream.h>

Die Klasse *ios* stellt Funktionen für Eingabe- und Ausgabe-Streams gemeinsam zur Verfügung. Die von ihr abgeleiteten Klassen (*istream, ostream* und *iostream*) implementieren formatierte Eingabe und Ausgabe. Die unmittelbaren Nachkommen sind *istream, ostream, fstreambase* und *strstreambase*.

Konstruktor `ios();`

Instantiiert *ios* ohne zugeordnetes *streambuf*-Objekt.

Konstruktor `ios(streambuf *);`

Verknüpft das gegebene *streambuf*-Objektmit dem Stream.

Daten-elemente

```
static const long adjustfield;
  // left / right / internal
static const long basefield;
  // dec / oct / hex
static const long floatfield;
  // scientific / fixed
streambuf        *bp;
  // das zugehörige streambuf-Objekt                    protected
int              x_fill;
  // Füllzeichen für die Ausgabe                        protected
long             x_flags;
  // Formatierungs-Flags                                protected
int              x_precision;
  // Gleitkomma-Genauigkeit für die Ausgabe             protected
int              state;
  // aktueller Status von streambuf                     protected
ostream          *x_tie;
  // der verbundene ostream, falls vorhanden            protected
int              x_width;
  // Feldbreite bei der Ausgabe                         protected
```

Element-funktionen

bad

```
int bad();
```

Liefert einen Wert ungleich Null zurück, wenn ein Fehler auftritt.

bitalloc

```
static long bitalloc();
```

Liefert einen Satz Flag-Bits. Der Rückgabewert kann für das Setzen, Löschen und Testen der Flags verwendet werden (wird für benutzerdefinierte Format-Flags verwendet).

clear

```
void clear(int = 0);
```

Setzt den Stream-Status auf den gegebenen Wert.

eof

```
int eof();
```

Liefert bei EOF einen Wert ungleich Null zurück.

fail

```
int fail();
```

Liefert einen Wert ungleich Null zurück, wenn eine Operation nicht ausgeführt werden konnte.

fill

```
char fill();
```

Liefert das aktuelle Füllzeichen zurück.

fill

```
char fill(char);
```

Setzt das Füllzeichen auf einen neuen Wert und liefert das vorhergehende zurück.

flags

```
long flags();
```

Liefert die aktuellen Format-Flags zurück.

flags

```
long flags(long);
```

Setzt die Format-Flags auf die gegebenen Werte; liefert die vorhergehenden Flags zurück. Mit *flags(0)* werden die Standardwerte gesetzt.

good

```
int good();
```

Liefert einen Wert ungleich Null zurück, wenn keine Status-Bits gesetzt sind (d.h. keine Fehler angezeigt wurden).

init

```
void init(streambuf *);                    protected
```

Führt die tatsächliche Initialisierung aus.

precision

```
int precision();
```

Liefert die aktuelle Gleitkommagenauigkeit.

precision
```
int precision(int);
```
Bestimmt die Gleitkommagenauigkeit; liefert die vorhergehende Einstellung zurück.

rdbuf
```
streambuf* rdbuf();
```
Liefert einen Zeiger auf die diesem Stream zugewiesene *streambuf*-Klasse zurück.

rdstate
```
int rdstate();
```
Liefert den Stream-Status zurück.

setf
```
long setf(long);
```
Setzt die Flags auf die gegebenen Werte; liefert die vorhergehenden Einstellungen zurück.

setf
```
long setf(long _setbits, long _field);
```
Die Bits, die den in *_field* markierten Bits entsprechen, werden gelöscht und dann durch die in *_setbits* vorhandenen ersetzt.

setstate
```
void setstate(int);
```
protected

Setzt alle Status-Bits.

sync_with_-stdio
```
static void sync_with_stdio();
```
Kombiniert *stdio*-Dateien mit *iostream*-Klassen. Diese Funktion sollte nicht für neuen Programmtext verwendet werden.

tie
```
ostream* tie();
```
Liefert den verbundenen Stream; wenn keiner vorhanden ist, so wird 0 zurückgegeben. »Verbunden« bedeutet: Wird ein Stream verwendet, ist der andere auf irgendeine Weise davon betroffen. Beispielsweise sind *cin* und *cout* miteinander verbunden; wird *cin* verwendet, leert es zunächst *cout*.

tie
```
ostream* tie(ostream*);
```
Verbindet einen anderen Stream mit diesem und liefert – falls vorhanden – den vorherigen verbundenen Stream zurück. Muß ein Eingabe-Stream Zeichen übernehmen oder benötigt ein Ausgabe-Stream weitere Zeichen, wird der verbundene Stream zuerst automatisch geleert. Per Voreinstellung sind *cin*, *cerr* und *clog* mit *cout* verbunden.

unsetf
```
long unsetf(long);
```
Löscht die Bits, die den in **long** markierten entsprechen; liefert die vorhergehenden Einstellungen zurück.

width
```
int width();
```
Liefert die aktuelle Einstellung der Breite zurück.

width `int width(int);`

Setzt die Breite wie angegeben; liefert die vorhergehende Breite zurück.

xalloc `static int xalloc();`

Liefert die nächste freie Position im Array, die für benutzerdefinierte Format-Flags verwendet werden kann.

iostream <iostream.h>

Diese von *istream* und *ostream* abgeleitete Klasse ist lediglich eine »Mischung« aus ihren Basisklassen, die sowohl Eingabe wie Ausgabe für einen Stream zuläßt. Sie dient als Basis für *fstream* und *strstream*.

Konstruktor `iostream(streambuf *);`

Verknüpft das gegebene *streambuf*-Objekt mit dem Stream.

iostream_withassign <iostream.h>

Diese Klasse ist eine *iostream*-Klasse mit einem zusätzlichen Zuweisungsoperator.

Konstruktor `iostream_withassign();`

Standardkonstruktor (ruft den Konstruktor von *iostream* auf).

Elementfunktionen Keine (obwohl der Operator = überladen ist).

istream <iostream.h>

Diese Klasse stellt formatierte und unformatierte Eingabe aus einer *streambuf*-Klasse zur Verfügung. Der Operator >> ist für alle vordefinierten Typen überladen, wie zu Beginn dieses Kapitels erläutert wurde. *istream* dient als Basis für *ifstream*, *iostream*, *istrstream* und *istream_withassign*.

Konstruktor `istream(streambuf *);`

Verknüpft das gegebene *streambuf*-Objekt mit dem Stream.

Elementfunktionen

gcount `int gcount();`

Liefert die Anzahl der zuletzt übernommenen Zeichen.

get

```
int get();
```

Übernimmt das nächste Zeichen oder EOF.

get

```
istream& get(signed char*, int len, char = '\n');
istream& get(unsigned char*, int    len, char = '\n');
```

Übernimmt Zeichen in den gegebenen Puffer **char***, bis ein Begrenzer (dritter Parameter) oder ein EOF auftritt, bzw. bis (*len*-1) Byte gelesen sind. Eine abschließende Null wird immer in den Ausgabe-String geschrieben, der Begrenzer nicht. Ein Fehler tritt nur dann auf, wenn keine Zeichen übernommen wurden.

get

```
istream& get(unsigned char&);
istream& get(signed char&);
```

Übernimmt ein einzelnes Zeichen in die gegebene Zeichenreferenz.

get

```
istream& get(streambuf&, char = '\n');
```

Übernimmt Zeichen in das gegebene *streambuf*-Objekt, bis der Begrenzer angetroffen wird.

getline

```
istream& getline(signed char *buffer, int, char = '\n');
istream& getline(unsigned char    *buffer, int,
                 char = '\n');
```

Wie *get*, außer, daß der Begrenzer ebenfalls übernommen wird. Der Begrenzer wird nicht in *buffer* kopiert.

ignore

```
istream& ignore(int n = 1, int delim= EOF);
```

Überspringt bis zu *n* Zeichen im Eingabe-Stream oder die Zeichen bis einschließlich *delim*.

peek

```
int peek();
```

Liefert das nächste Zeichen, ohne es zu übernehmen.

putback

```
istream& putback(char);
```

Schreibt ein Zeichen in den Stream zurück.

read

```
istream& read(signed char*, int);
istream& read(unsigned char*, int);
```

Übernimmt eine gegebene Anzahl von Zeichen in ein Array. Wenn ein Fehler auftritt, erhalten Sie mit *gcount*() die Anzahl der tatsächlich übernommenen Zeichen.

seekg

```
istream& seekg(long);
```

Positioniert auf eine absolute Position (wie sie von *tellg* zurückgeliefert wird).

seekg `istream& seekg(long, seek_dir);`

Positioniert auf eine Position relativ zur über *seek_dir* angegebenen Position. *seek_dir* ist wie folgt definiert:

```
enum seek_dir {beg, cur, end};
```

Das entspricht dem Anfang, der aktuellen Position und dem Ende des Streams.

tellg `long tellg();`

Liefert die aktuelle Stream-Position.

istream_withassign <iostream.h>

Diese Klasse ist eine *istream*-Klasse mit einem zusätzlichen Zuweisungsoperator.

Konstruktor `istream_withassign();`

Standard-Konstruktor (ruft den Konstruktor von *istream* auf).

Element-funktionen Keine (obwohl der Operator = überladen ist).

istrstream <strstrea.h>

Stellt Eingabeoperationen für eine *strstreambuf*-Klasse zur Verfügung. Die Klasse ist von *strstreambase* und *istream* abgeleitet.

Konstruktor `istrstream(const char *);`

Erzeugt eine *istrstream*-Klasse mit dem angegebenen String (ein Null-Zeichen wird niemals übernommen).

Konstruktor `istrstream(const char *, int n);`

Erzeugt eine *istrstream*-Klasse unter Verwendung von maximal *n* Bytes des angegebenen Strings.

ofstream <fstream.h>

Stellt Eingabe-Operationen für ein *filebuf*-Objekt zur Verfügung und ist von *fstreambase* und *ostream* abgeleitet.

Konstruktor `ofstream();`

Erzeugt ein *ofstream*-Objekt, das mit keiner Datei verbunden ist.

Konstruktor
```
ofstream(const char*, int = ios::out,
int = filebuf::openprot);
```
Erzeugt ein *ofstream*-Objekt, öffnet eine Datei und verbindet sie mit *ofstream*.

Konstruktor
```
ofstream(int);
```
Erzeugt ein *ofstream*-Objekt und verbindet es mit Hilfe des Deskriptors mit einer geöffneten Datei.

Konstruktor
```
ofstream(int fd, char*, int);
```
Erzeugt ein *ofstream*-Objekt, das mit einer geöffneten Datei verbunden ist und verwendet den angegebenen Puffer.

Element-funktionen

open
```
void open(const char*, int, int = filebuf::openprot);
```
Öffnet eine Datei für ein *ofstream*-Objekt.

rdbuf
```
filebuf* rdbuf();
```
Liefert den verwendeten Puffer zurück.

ostream <iostream.h>

Diese Klasse ergänzt *streambuf* um formatierte und unformatierte Ausgabe. Der Operator << ist für alle vordefinierten Datentypen überladen. (Eine Erläuterung dazu finden Sie im Abschnitt »Ausgabe« weiter vorne in diesem Kapitel.) Die Klasse dient als Basis für *constream, iostream, ofstream, ostrstream* und *ostream_withassign*

Konstruktor
```
ostream(streambuf *);
```
Verknüpft das gegebene *streambuf*-Objekt mit dem Stream.

Element-funktionen

flush
```
ostream& flush();
```
Leert den Stream.

put
```
ostream& put(char);
```
Fügt das Zeichen ein.

seekp
```
ostream& seekp(long);
```
Positioniert auf eine absolute Position (wie sie von *tellp* zurückgeliefert wird).

seekp
```
ostream& seekp(long, seek_dir);
```
Positioniert auf eine Position relativ zur über *seek_dir* angegebenen Position. *seek_dir* ist wie folgt definiert:
```
enum seek_dir {beg, cur, end};
```

tellp
```
long tellp();
```
Liefert die aktuelle Stream-Position.

write
```
ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int      n)
```
Fügt *n* Zeichen (auch Null-Zeichen) ein.

ostream_withassign <iostream.h>

Diese Klasse ist eine *ostream_withassign*-Klasse mit einem zusätzlichen Zuweisungsoperator.

Konstruktor
```
ostream_withassign();
```
Null-Konstruktor (ruft den Konstruktor von *ostream* auf).

Element-funktionen Keine (obwohl der Operator = überladen ist).

ostrstream <strstrea.h>

Stellt Ausgabeoperationen für eine *strstreambuf*-Klasse zur Verfügung. Die Klasse ist von *strstreambase* und *ostream* abgeleitet.

Konstruktor
```
ostrstream();
```
Erzeugt ein dynamisches *ostrstream*-Objekt.

Konstruktor
```
ostrstream(char*, int, int = ios::out);
```
Erzeugt ein *ostrsteam*-Objekt mit dem spezifizierten *n* Byte großen Puffer. Wenn *mode* gleich *ios::app* oder *ios::ate* ist, dann wird der Eingabe-/Ausgabezeiger auf das Null-Zeichen des Strings positioniert.

Element-funktionen

pcount
```
char *pcount();
```
Liefert die Anzahl der aktuell im Puffer gespeicherten Bytes.

str
```
char *str();
```
Liefert den Puffer zurück und verhindert Änderungen an ihm. Wenn der Puffer dynamisch war, müssen Sie ihn freigeben.

streambuf <iostream.h>

Dies ist eine Klasse zur Bearbeitung von Puffern. Ihre Anwendungen werden auf Puffer und Pufferfunktionen über einen Zeiger auf *streambuf* zugreifen, der von *ios* gesetzt wird. Die Klasse dient als Basis für *filebuf* und *strstreambuf*.

Konstruktor `streambuf();`

Erzeugt ein leeres Pufferobjekt.

Konstruktor `streambuf(char*, int);`

Verwendet das gegebene Array und die gegebene Größe für den Puffer.

Elementfunktionen

allocate `int allocate();` **protected**

Bestimmt einen Pufferbereich.

base `char *base();` **protected**

Liefert den Beginn des Pufferbereichs.

blen `int blen();` **protected**

Liefert die Länge des Pufferbereichs.

eback `char *eback();` **protected**

Liefert die Größe des Zurückgeben-Bereichs in einem Eingabebereich.

ebuf `char *ebuf();` **protected**

Liefert Ende +1 für den Pufferbereich.

egptr `char *egptr();` **protected**

Liefert Ende +1 für den Eingabebereich.

epptr `char *epptr();` **protected**

Liefert Ende +1 für den Ausgabebereich.

gbump `void gbump(int);` **protected**

Bewegt den Eingabezeiger weiter.

gptr `char *gptr();` **protected**

Liefert die nächste Position im Eingabebereich.

in_avail `int in_avail();`

Liefert die Anzahl der im Eingabepuffer verbleibenden Zeichen.

out_waiting
```
int out_waiting();
```
Liefert die Anzahl der im Ausgabepuffer verbleibenden Zeichen.

pbase
```
char *pbase();
```
protected

Liefert den Beginn des Ausgabebereichs.

pbump
```
void pbump();
```
protected

Bewegt den Eingabezeiger weiter.

pptr
```
char *pptr();
```
protected

Liefert die nächste Position im Ausgabebereich.

sbumpc
```
int sbumpc();
```
Liefert das aktuelle Zeichen aus dem Eingabepuffer und positioniert auf das nächste Zeichen.

seekoff
```
virtual streampos seekoff(streamoff, ios::seek_dir,
int = (ios::in | ios::out);
```
Verschiebt den Eingabe- oder Ausgabezeiger (das dritte Argument legt fest, welcher Zeiger oder ob beide verwendet werden sollen) relativ zur über *seek_dir* angegebenen Position.

seekpos
```
virtual streampos seekpos(streampos, int = (ios::in |
ios::out);
```
Verschiebt den Eingabe- oder Ausgabezeiger an eine absolute Position.

setb
```
void setb(char *, char *, int = 0);
```
protected

Setzt den Pufferbereich.

setbuf
```
virtual streambuf* setbuf(signed char *, int);
streambuf* setbuf(unsigned char *, int);
```
Stellt eine Verbindung zu einem gegebenen Puffer her.

setg
```
void setg(char *, char *, char *)
```
protected

Initialisiert die Eingabezeiger.

setp
```
void setp(char *, char *);
```
protected

Initialisiert die Ausgabezeiger.

sgetc
```
int sgetc();
```
Liefert das nächste Zeichen im Eingabepuffer, ohne es zu übernehmen.

sgetn
```
int sgetn(char*, int n);
```
Holt die nächsten *n* Zeichen aus dem Eingabepuffer.

snextc `int snextc();`

Positioniert auf das nächste Zeichen im Eingabepuffer und liefert es zurück.

sputback `int sputbackc(char);`

Gibt ein Zeichen an die Eingabe zurück.

sputc `int sputc(int);`

Übergibt ein Zeichen an den Ausgabepuffer.

sputn `int sputn(const char*, int n);`

Übergibt *n* Zeichen an den Ausgabepuffer.

stossc `void stossc();`

Positioniert auf das nächste Zeichen im Eingabepuffer.

unbuffered `int unbuffered(int);` **protected**

Bestimmt den Status des Puffers.

unbuffered `int unbuffered();` **protected**

Liefert einen Wert, der ungleich Null ist, wenn nicht gepuffert wird.

strstreambase <strstrea.h>

Spezialisiert die *ios*-Klasse auf String-Streams. Mit Ausnahme der Elementfunktion *rdbuf()* sind alle Elementfunktionen vom Typ **protected**. Die Klasse dient *strstream, istrstream* und *ostrstream* als Basis.

Konstruktor `strstreambase();` **protected**

Erzeugt ein leeres *strstreambase*-Objekt.

Konstruktor `strstreambase(const char *, int, char *start);`

Erzeugt ein *strstreambase*-Objekt mit dem angegebenen Puffer und der angegebenen Startposition.

Element-funktionen

rdbuf `strstreambuf * rdbuf();`

Liefert einen Zeiger auf das mit diesem Objekt verknüpfte *strstreambuf*-Objekt.

strstreambuf <strstrea.h>

Spezialisiert die *streambuf*-Klasse für Formatierung im Speicher.

Konstruktor
```
strstreambuf();
```
Erzeugt ein *strstreambuf*-Objekt. Speicher wird je nach Erfordernis dynamisch reserviert.

Konstruktor
```
strstreambuf(void * (*) (long), void * (*) (void *));
```
Erzeugt einen dynamischen Puffer mit den angegebenen Funktionen zur Reservierung und Freigabe von Speicher.

Konstruktor
```
strstreambuf(int n);
```
Erzeugt ein *strstreambuf*-Objekt und reserviert zu Beginn einen dynamischen Puffer mit mindestens *n* Bytes.

Konstruktor
```
strstreambuf(signed char *, int, signed char *end = 0);
strstreambuf(unsigned char *, int,
unsigned char *end = 0);
```
Erzeugt ein *strstreambuf*-Objekt mit dem angegebenen statischen Puffer. Wenn *end* nicht Null ist, dient es als Begrenzer für den Puffer.

Elementfunktionen

doallocate
```
virtual int doallocate();
```
Führt Low-Level-Reservierungen für Puffer durch.

freeze
```
void freeze(int = 1);
```
Wenn der Eingabeparameter nicht Null ist, wird das Speichern von Zeichen im Puffer verhindert. Durch Übergabe einer Null kann das Speichern wieder zugelassen werden.

overflow
```
virtual int overflow(int = EOF);
```
Leert einen Puffer durch Schreiben in sein Ziel. Jede abgeleitete Klasse sollte die erforderlichen Aktionen definieren.

seekoff
```
virtual streampos seekoff(streamoff,
ios::seek_dir, int);
```
Verschiebt den Zeiger relativ zur aktuellen Position.

setbuf
```
virtual streambuf* setbuf( char*, int);
```
Bestimmt den zu verwendenden Puffer.

str
```
char *str();
```
Liefert einen Zeiger auf den Puffer zurück und verhindert Änderungen an ihm.

underflow

```
virtual int underflow();
```

Macht die Eingabe verfügbar. Diese Funktion wird aufgerufen, wenn ein Zeichen erforderlich und **strstreambuf** leer ist. Jede abgeleitete Klasse sollte die erforderlichen Aktionen definieren.

strstream <strstrea.h>

Stellt gleichzeitige Ein- und Ausgabe für eine *strstreambuf*-Klasse zur Verfügung. Diese Klasse ist von *strstreambase* und *iostream* abgeleitet.

Konstruktor

```
strstream();
```

Erzeugt ein *strstream*-Objekt.

Konstruktor

```
strstream(char*, int n, int mode);
```

Erzeugt ein *strstream*-Objekt mit dem angegebenen Puffer, der eine Größe von *n* Byte hat. Wenn *mode* gleich *ios::app* oder *ios::ate* ist, wird der Eingabe/Ausgabezeiger auf das Null-Zeichen des Strings positioniert.

Elementfunktionen

str

```
char *str();
```

Liefert den Puffer zurück und verhindert Änderungen an ihm. Wenn der Puffer dynamisch war, müssen Sie ihn freigeben.

6. Arithmetik

In diesem Kapitel werden die Gleitkomma-Optionen vorgestellt und der Gebrauch der komplexen Arithmetik erläutert.

Gleitkomma-Optionen

In C werden zwei Arten von Zahlen unterschieden: Integerzahlen (**int**, **short**, **long** usw.) und Gleitkommazahlen (**float**, **double** und **long double**).

Der Prozessor Ihres Rechners ist für eine effektive Verarbeitung von Integerwerten eingestellt. Die Verarbeitung von Gleitkommawerten erfordert jedoch mehr Zeit und Aufwand.

Die Prozessorfamilie *iAPx86* hat eine entsprechende Familie von mathematischen Koprozessoren, die 8087-, 80287- und 80387-Prozessoren. Wir bezeichnen die gesamte Familie als 80x87 oder einfach als »Koprozessor«. Bei 80486-Prozessoren ist der mathematische Koprozessor bereits integriert (Ausnahme: 80486sx).

Der 80x87 ist ein mathematischer Hardware-Prozessor, der in Ihren PC eingebaut werden kann. Er führt Gleitkommabefehle sehr schnell aus. Wenn Sie sehr viel mit Gleitkommabefehlen arbeiten, werden Sie wahrscheinlich einen Koprozessor haben wollen. Die CPU in Ihrem Computer stellt eine Schnittstelle zum 80x87 über spezielle Hardware-Leitungen her.

Die Emulation des 80x87

In Turbo C++ ist *Emulation* für die Option *Code-Generierung* voreingestellt. Diese Option kann bei Programmen eingesetzt werden, die mit oder ohne Gleitkommazahlen arbeiten und auf Computern mit oder ohne mathematischen Koprozessoren laufen sollen.

Mit der Option *Emulation* erzeugt der Compiler ein Programm so, als ob ein Koprozessor vorhanden wäre, erzeugt aber auch die Informationen (FIXUPS), die Windows für die Durchführung Ihres Programms ohne Koprozessor benötigt.

Bei der Ausführung des Programms wird der Koprozessor verwendet, falls einer vorhanden ist; steht zur Laufzeit kein Koprozessor zur Verfügung, setzt das Programm spezielle Software ein, die den 80x87 *emuliert*.

Verwendung des 80x87-Codes

Wenn Ihr Programm nur auf einem Computer laufen soll, der mit einem 80x87-Koprozessor ausgestattet ist, können Sie in Ihren .EXE-Dateien bis zu 10 KBytes einsparen, weil dann die Überprüfung auf Vorhandensein eines 80x87 und die Emulation des 80x87 entfällt.

Wählen Sie einfach das Schaltfeld *80287* im Dialogfenster *Erweiterte Code-Generierung*. Turbo C++ linkt dann Ihre Programme mit FP87.LIB anstatt mit EMU.LIB.

Falls Sie keine Gleitkomma-Operationen brauchen

Wenn Ihr Programm keinen Gleitkomma-Code enthält, können Sie etwas Zeit beim Linken einsparen, indem Sie die Option *Nicht Verwenden* wählen. In diesem Fall führt Turbo C++ den Linkvorgang ohne die Bibliothek MATH*x*.LIB durch.

Beschleunigte Gleitkommaberechnungen

Turbo C++ verfügt über eine Option zur Beschleunigung von Gleitkommaberechnungen. Mit Hilfe dieser Option können gewisse Optimierungen durchgeführt werden, die im Gegensatz zur korrekten C-Semantik stehen.

Dazu ein Beispiel:

```
double x;
x = (float) (3.5*x);
```

Bei einer korrekten Ausführung wird *x* mit 3.5 multipliziert. Der resultierende **long double**-Wert wird auf **float**-Genauigkeit abgeschnitten und anschließend als **double** in *x* gespeichert. Beim Einsatz der Option wird das **long double**-Produkt direkt in ein **double**-Typ umgewandelt.

Diese Option ist in Turbo C++ voreingestellt, da sich nur wenige Programme auf den Verlust an Genauigkeit verlassen, der sich aus der vorübergehenden Zuweisung zu einem kürzeren **float** ergeben.

Register und Koprozessor

Bei der Verwendung von Registern und Gleitkomma-Code gibt es einige Punkte, die Sie bedenken sollten:

1. Im 80x87-Emulationsmodus werden »wraparound« von Registern und bestimmte andere 80x87-Eigenschaften nicht unterstützt.

2. Wenn Sie Gleitkommabefehle zusammen mit Inline-Assembler verwenden, müssen Sie ganz besonders auf die Register achten. Sie müssen die Werte der 80x87-Register vor dem Aufruf von Funktionen, die den Koprozessor verwenden, ermitteln und sichern – außer Sie sind sich sicher, daß genügend freie Register vorhanden sind.

Fehlerbedingungen bei Gleitkomma-Operationen

Turbo C++ bricht per Voreinstellung Programme ab, wenn ein Gleitkomma-Überlauf oder eine Division durch Null auftreten.

Derartige Fehlerbedingungen können durch einen Aufruf von *_control87* in *main* vor der Ausführung von Gleitkomma-Operationen maskiert und abgefangen werden. Betrachten Sie folgendes Beispiel:

```
#include <float.h>
main() {
   _control87 (MCW_EM,MCW_EM);
   ...
}
```

Durch den Aufruf von *_status87* oder *_clear87* können Sie feststellen, ob eine Fehlerbedingung aufgetreten ist. Eine detaillierte Erläuterung dieser Funktionen finden Sie in der Online-Hilfe.

Bestimmte arithmetische Fehler können auch in Bibliotheksfunktionen vorkommen, z. B. Quadratwurzel aus einer negativen Zahl. Normalerweise hat das eine Fehlermeldung zur Folge und es wird NAN (not-a-number) zurückgeliefert. Durch die Verwendung von NAN ergibt sich später höchstwahrscheinlich eine Gleitkomma-Fehlerbedingung, die zum Programmabbruch führt, wenn sie nicht abgefangen wird. Sie können die Ausgabe der Fehlermeldung verhindern, indem Sie folgende *matherr*-Version in Ihr Programm einfügen:

```
#include <math.h>
int cdecl matherr(struct exception *e)
{
   return 1;            /* Der Fehler wurde abgefangen */
}
```

Der Einsatz von *matherr* zur Behandlung anderer arithmetischer Fehler wird nicht empfohlen, da er nicht mehr gebräuchlich ist und in späteren Versionen von Turbo C++ möglicherweise nicht mehr unterstützt wird.

Komplexe Arithmetik

Komplexe Zahlen sind Zahlen der Form $x + yi$, wobei x und y reelle Zahlen sind, und i die imaginäre Quadratwurzel von -1 ist. In Turbo C ++ war schon immer folgender Typ in math.h definiert:

```
struct complex
{
   double x, y;
};
```

Dieser Typ eignet sich zur Aufnahme komplexer Zahlen, die als Paare von reellen Zahlen betrachtet werden können. In C ist die Arithmetik mit komplexen Zahlen sehr mühsam – in C++ ist dies wesentlich einfacher. Eine ausführliche Besprechung der Klasse *complex* finden Sie in der Online-Hilfe.

Wenn Sie in C++ mit komplexen Zahlen arbeiten möchten, müssen Sie dazu nur complex.h – mit *include* – verwenden. In complex.h sind folgende Operatoren und Funktionen überladen:

- alle gebräuchlichen arithmetischen Operatoren,
- die Stream-Operatoren >> und <<,
- die normalen *math*-Funktionen, wie *sqrt* und *log*.

Die *complex*-Bibliothek wird nur dann aufgerufen, wenn das Argument vom Typ *complex* ist. Die komplexe Quadratwurzel von -1 erhalten Sie daher mit

```
sqrt(complex(-1))
```

und nicht mit

```
sqrt(-1)
```

Das folgende Beispiel zeigt die Verwendung von komplexen Zahlen anhand einer Funktion, die eine diskrete Fourier-Transformation berechnet.

```
#include <complex.h>
// berechnet eine diskrete Fourier-Transformation
// von a[0],...,a[n-1].
void Fourier(int n, complex a[], complex b[])
{
   int j, k;
   complex i(0,1);    // Quadratwurzel von -1
   for (j = 0; j < n; ++j)
   {
      b[j] = 0;
      for (k = 0; k < n; ++k)
         b[j] += a[k] * exp(2*M_PI*j*k*i/n);
      b[j] /= sqrt(n);
   }
}
```

Die BCD-Arithmetik

Die Arithmetik von Turbo C++ arbeitet (wie fast alle Computer und Compiler) mit binären Zahlen (Basis 2). Dies ist manchmal verwirrend, wenn Sie nur die dezimale Darstellung (Basis 10) gewöhnt sind. Viele Zahlen, die im Dezimalsystem (Basis 10) exakt dargestellt werden, können mit binären Zahlen nur annähernd wiedergegeben werden (zum Beispiel 0.01).

Binäre Zahlen sind bei den meisten Anwendungen vorzuziehen, aber in einigen Fällen ist der Rundungsfehler, der bei der Umwandlung von Dezimal- in Binärzahlen entsteht, nicht tolerierbar. Dies trifft vor allem bei Finanz- oder Lohnbuchhaltungsprogrammen zu. Ein Beispiel soll dies verdeutlichen: das folgende Programm addiert 100 Pfennige auf und subtrahiert dann eine Mark:

```
#include <stdio.h>
int i;
float x = 0.0;
for (i = 0; i < 100; ++i)
   x += 0.01;
x -= 1.0;
printf("100*.01 - 1 = %g\n",x);
```

Die korrekte Ergebnis sollte 0.0 sein, das Programm berechnet aber eine sehr kleine Zahl, die nur annähernd 0.0 ist.

Die Berechnung vergrößert den winzigen Rundungsfehler, der bei der Umwandlung von 0.01 zur Basis 2 entsteht. Wird der Typ von *x* in den Typ **double** oder **long double** geändert, reduziert sich zwar der Fehler, er ist aber nicht eliminiert.

Zur Lösung dieses Problems bietet sich der C++ Typ *bcd* an, der in bcd.h deklariert ist. Mit *bcd* wird die Zahl 0.01 korrekt repräsentiert und die *bcd*-Variable *x* liefert den korrekten Pfennigbetrag:

```
#include <bcd.h>
   int i;
   bcd x = 0.0;

   for (i = 0; i < 100; ++i)
   x += 0.01;
   x -= 1.0;

   cout << "100*.01 -1 = " << x << "\n";
}
```

Folgendes sollte beim Einsatz von *bcd* beachtet werden:

- *bcd* eliminiert nicht alle Rundungsfehler; bei Berechnungen wie 1.0/3.0 bleibt ein Rundungsfehler bestehen.
- Die gebräuchlichen *math*-Funktionen, wie *sqrt* und *log*, sind für *bcd*-Argumente überladen.
- BCD-Zahlen haben eine Genauigkeit von ca. 17 Dezimalstellen und umfassen einen Bereich zwischen 1×10^{-125} und 1×10^{125}.

Konvertierung von BCD-Zahlen

bcd ist ein vom Typ **float**, **double** oder **long double** verschiedener Datentyp; BCD-Arithmetik wird nur durchgeführt, wenn mindestens ein Operand vom Typ *bcd* ist.

Wichtig: Mit der *bcd*-Elementfunktion *real* kann eine *bcd*-Zahl in das Format zur Basis 2 (**float**, **double** oder **long double**) zurückkonvertiert werden. Die Konvertierung erfolgt allerdings nicht automatisch. *real* sorgt für die Konvertierung in den Typ **long double**, der dann mit den üblichen C-Konvertierungen in einen anderen Typ konvertiert werden kann.

Zum Beispiel kann

```
bcd a = 12.1;
```

mit einer der folgenden vier Programmzeilen ausgedruckt werden:

```
double x = a; printf("a = %g", x);
printf ("a = %Lg", real(a));
printf ("a = %g", (double)real(a));
cout << "a = " << a;
```

Da *printf* keine Überprüfung der Argumente durchführt, muß zum Format-Spezifizierer das *L* angegeben werden, wenn der **long double**-Wert *real*(*a*) übergeben wird.

Anzahl der Dezimalstellen

Sie können angeben, wieviele Stellen hinter dem Dezimalpunkt bei der Konvertiertung einer binären Zahl in einen *bcd*-Typ übernommen werden sollen. Die Anzahl der Stellen ist ein optionales zweites Argument zum Konstruktor *bcd*. Betrachten Sie dazu folgendes Beispiel. Mit

```
bcd a = bcd(1000.00/7, 2)
```

kann der Betrag DM 1000.00/7 in eine *bcd*-Variable konvertiert werden, wobei auf den nächsten Pfennig gerundet wird. Die Ziffer 2 legt fest, daß zwei Dezimalstellen übernommen werden.

Daraus ergibt sich:

```
1000.00/7          =  142.85714...
bcd(1000.00/7, 2)  =  142.860
bcd(1000.00/7, 1)  =  142.900
bcd(1000.00/7, 0)  =  143.000
bcd(1000.00/7, -1) =  140.000
bcd(1000.00/7, -2) =  100.000
```

Die Rundungsmethode wird durch die IEEE genormt und entspricht dem Verfahren der kaufmännischen Rechnung: Es wird auf die nächste ganze Zahl – im Zweifelsfall auf eine gerade Zahl – gerundet:

```
bcd(12.335, 2)     =  12.34
bcd(12.345, 2)     =  12.34
bcd(12.355, 2)     =  12.36
```

7. BASM und Inline-Assembler

In diesem Kapitel wird erläutert, wie Sie mit Hilfe des integrierten Inline-Assemblers (BASM) von Turbo C++ Assembler-Routinen in Ihre C- und C++ Programme einbinden können. Dazu ist kein separater Assembler erforderlich. Derartiger Assembler-Code wird als *Inline-Assemblierung* bezeichnet, weil er zusammen mit C-Routinen und nicht separat compiliert wird. Das Ergebnis wird mit den vom C-Compiler erzeugten Modulen gelinkt.

Turbo C++ unterstützt das traditionelle Mischen von Sprachen, bei dem Ihr C-Programm Assembler-Routinen aufruft (oder umgekehrt), die getrennt voneinander von TASM (Turbo Assembler) erzeugt wurden. Um C und Maschinencode korrekt kombinieren zu können, müssen Sie wissen, wie 8086-Maschinencode geschrieben wird und wie Segmente, Datenkonstanten, etc. definiert werden. Außerdem müssen Ihnen die *Aufruf-Konventionen* (d.h. die Abfolge der Parameterübergabe) in C und Assembler vertraut sein (dazu gehört auch die Parameterübergabe des Typs **pascal** in C).

Inline-Assemblierung

Turbo C++ erlaubt es Ihnen, Maschinenbefehle direkt in C- oder C++ Programme zu schreiben. Dieses Verfahren wird als *Inline-Assemblierung* bezeichnet.

BASM

Mit Hilfe des integrierten Assemblers (BASM) kann Turbo C++ Ihre Maschinenbefehle auch dann assemblieren, wenn Sie keine der oben aufgeführten Methoden zu Aktivierung von TASM verwenden. Bei der Verwendung von BASM gelten allerdings die folgenden Einschränkungen:

- Assembler-Makros können nicht verwendet werden.
- BASM kann keine 80386- und 80486-Befehle verarbeiten.
- Ideal-Mode-Syntax ist nicht möglich.
- Es kann nur eine begrenzte Anzahl von Assembler-Direktiven verwendet werden (vgl. den Abschnitt »Assembler-Direktiven« weiter hinten).

Inline-Syntax

Wenn Sie in Ihrem C-Programm Maschinenbefehle verwenden wollen, sollten Sie mit den Prozessorbefehlen und der Architektur des 8086 vertraut sein. Außerdem sollten Sie wissen, wie und wo die verwendeten Befehle angewandt werden müssen und was sie bewirken. Ein Befehl in Maschinensprache wird innerhalb eines C-Programms mit dem Schlüsselwort **asm** kenntlich gemacht. Das Format dafür sieht folgendermaßen aus:

asm *opcode operanden* ; oder *Zeilenvorschub*

- *opcode* ist ein gültiger 8086-Befehl (in der Tabelle 7.1 sind alle zulässigen Opcodes aufgeführt).
- *operanden* steht für den oder die Operanden, auf die sich *opcode* bezieht. Operanden können auch Konstanten, Variablen oder Label des C-Programms sein.
- Ein Strichpunkt (;) oder ein Zeilenvorschub kennzeichnen das Ende der **asm**-Anweisung.

Mehrere **asm**-Anweisungen in einer Zeile sind zulässig, solange sie durch Strichpunkte voneinander getrennt sind – nicht erlaubt ist dagegen die Fortsetzung einer **asm**-Anweisung in der nächsten Zeile.

Falls Sie eine Reihe von **asm**-Anweisungen verwenden, können diese in geschweiften Klammern stehen:

```
asm {
    pop ax; pop ds
    iret
}
```

Die offene geschweifte Klammer *muß* sich in der gleichen Zeile befinden, wie das Schlüsselwort **asm**. Strichpunkte werden – im Gegensatz zu TASM – *nicht* als Einleitung eines Kommentars erkannt. Verwenden Sie zur Kommentierung von **asm**-Anweisungen Kommentare im Stil von C:

```
asm mov ax,ds;                /* Dieser Kommentar ist o.k. */
asm {pop ax; pop ds; iret;}   /* ebenfalls erlaubt */
asm push ds                   ;DIESER KOMMENTAR IST KEINER!
```

Der Assembler-Teil der Anweisung wird direkt an die Ausgabe weitergegeben und in die Maschinensprache eingebettet, die Turbo C++ aus den C- oder C++ Befehlen erzeugt. C-Symbole werden durch die entsprechenden Äquivalente in Maschinensprache ersetzt.

Beachten Sie, daß bei der Inline-Assemblierung kein kompletter Assembler zur Verfügung steht und deshalb bestimmte Konstrukte in Maschinensprache nicht akzeptiert werden. In derartigen Fällen erhalten Sie von Turbo C++ eine Fehlermeldung. Sie können dann entweder Ihren Inline-Code vereinfachen, so daß ihn der Assembler akzeptiert, oder Sie können TASM aufrufen. Im letzteren Fall werden alle Fehler von TASM behandelt. Die Position der Fehler kann TASM allerdings nicht feststellen, da die Nummer der ursprünglichen C-Quelltextzeile verlorengeht.

Jede **asm**-Anweisung wird als eine C-Anweisung gezählt:

```
myfunc()
{
   int i;
   int x;
   if (i > 0)
     asm  mov  x,4
   else
     i = 7;
}
```

Dieses Konstrukt ist eine zulässige **if**-Anweisung in C. Nach dem Befehl *mov x,4* wird kein Strichpunkt benötigt. **asm**-Anweisungen sind die einzigen Anweisungen in C, die nicht mit Strichpunkten enden müssen. Das Ende der **asm**-Anweisung wird durch den Beginn einer neuen Zeile angezeigt. Das ist zwar nicht hundertprozentig portierbar, wird aber von einigen weitverbreiteten UNIX-Compilern genauso gehandhabt.

Eine **asm**-Anweisung kann als ausführbarer Befehl Teil einer Funktion sein oder als externe Deklaration außerhalb sämtlicher Funktionen geschrieben werden. Im ersten Fall speichert der Compiler den Befehl zusammen mit dem restlichen Code der Funktion im Code-Segment, ansonsten werden die entsprechenden Bytes in das Datensegment gesetzt.

Opcodes

Sämtliche Opcodes des 8086 können als Inline-Assemblierungsanweisungen eingesetzt werden. Der Compiler von Turbo C++ erlaubt vier Klassen von Befehlen:

- Normale Befehle – die regulären Opcodes des 8086.
- Stringbefehle – spezielle Befehle zur Stringbehandlung.
- Sprungbefehle – diverse Sprung-Opcodes.
- Assembler-Direktiven für die Zuweisung und Definition von Daten.

Die Prüfung der Operanden erfolgt nicht durch den Compiler, sondern durch den Assembler. Dasselbe gilt für das Format von Operanden. Folgende Kürzel für Opcodes können bei der Inline-Assemblierung verwendet werden:

Tabelle 7.1 Kürzel für Opcodes

aaa	bound	cmp	f2xm1
aad	call	cwd	fabs
aam	cbw	daa	fadd
aas	clc	das	faddp
adc	cld	dec	fbld
add	cli	div	fbstp
and	cmc	enter	fchs

fclex	fldе2t	fsubp	out
fcom	fldlg2	fsubr	pop
fcomp	fldln2	fsubrp	popa
fcompp	fldpi	ftst	popf
fdecstp**	fldz	fwait	push
fdisi	fmul	fxam	pusha
fdiv	fmulp	fxch	pushf
fdivp	fnclex	fxtract	rcl
fdivr	fndisi	fyl2x	rcr
fdivrp	fneni	fyl2xp1	ret
feni	fninit	hlt	rol
ffree**	fnop	idiv	ror
fiadd	fnsave	imul	sahf
ficom	fnstcw	in	sal
ficomp	fnstenv	inc	sar
fidiv	fnstsw	int	sbb
fidivr	fpatan	into	shl
fild	fprem	iret	shr
fimul	fptan	lahf	smsw
fincstp**	frndint	lds	stc
finit	frstor	lea	std
fist	fsave	leave	sti
fistp	fscale	les	sub
fisub	fsqrt	lsl	test
fisubr	fst	mov	verr
fld	fstcw	mul	verw
fld1	fstenv	neg	wait
fldcw	fstp	nop	xchg
fldenv	fstsw	not	xlat
fldl2e	fsub	or	xor

Die mit ** gekennzeichneten Opcodes werden nicht unterstützt, wenn sie in Routinen vorkommen, die von der Gleitkomma-Emulation Gebrauch machen.

Stringbefehle

Neben den oben aufgeführten Opcodes können auch die Befehle in der Tabelle 7.2 verwendet werden (allein oder mit Wiederholungspräfixen):

Tabelle 7.2 Stringbefehle

cmps	insw	movsb	outsw	stos
cmpsb	lods	movsw	scas	stosb
cmpsw	lodsb	outs	scasb	stosw
ins	lodsw	outsb	scasw	
insb	movs			

Präfixe
Die folgenden Präfixe können benutzt werden: *lock*, *rep*, *repe*, *repne*, *repnz* und *repz*.

Sprungbefehle
Sprungbefehle werden auf spezielle Weise behandelt: Da ein Label nicht als Teil einer **asm**-Anweisung definiert werden kann, müssen normale C-Labels angesprungen werden (siehe »Sprungbefehle und Labels« auf den nächsten Seiten). Tabelle 7.3 listet die zulässigen Sprungbefehle auf:

Tabelle 7.3 Sprungbefehle

ja	jge	jnc	jns	loop
jae	jl	jne	jnz	loope
jb	jle	jng	jo	loopne
jbe	jmp	jnge	jp	loopnz
jc	jna	jnl	jpe	loopz
jcxz	jnae	jnle	jpo	
je	jnb	jno	js	
jg	jnbe	jnp	jz	

Assembler-Direktiven
Turbo C++ erlaubt folgende Assembler-Direktiven innerhalb von Inline-Assemblierungsanweisungen: *db*, *dd*, *dw* und *extrn*.

Der Zugriff auf Daten und Funktionen

In **asm**-Anweisungen können C-Bezeichner verwendet werden. Turbo C++ wandelt sie automatisch in die entsprechenden Operanden für den Assembler um und stellt Unterstriche voran. Sie können jeden Bezeichner einschließlich automatischer (lokaler) Variablen, Registervariablen und Funktionsparameter verwenden.

Im allgemeinen kann jeder C-Bezeichner anstelle eines Adreßoperanden erscheinen. Variablen der Speicherklasse **register** können überall dort eingesetzt werden, wo ein Register als Operand zulässig wäre.

Trifft der Assembler während des Parsens der Operanden in einer **asm**-Anweisung auf einen Bezeichner, sucht er diesen in der Symboltabelle von C. Die Namen des 8086-Registers werden dabei nicht miteinbezogen. Registernamen können sowohl mit Groß- als auch mit Kleinbuchstaben geschrieben werden.

Inline-Assemblierung und Registervariablen
Innerhalb von **asm**-Anweisungen können die Register *SI* und *DI* problemlos für Hilfsaufgaben genutzt werden. Der Compiler benutzt in diesem Fall *SI* und *DI* nicht für Registervariablen.

Inline-Assemblierung, Offsets und Variablengrößen

Um die exakten Offsets von lokalen Variablen brauchen Sie sich nicht selbst zu kümmern – es reicht die Angabe des entsprechenden C-Bezeichners.

Es ist jedoch notwendig, die entsprechenden WORD PTR, BYTE PTR oder andere Größenangaben in die Assembler-Anweisung einzubinden. Beispiele dafür sind der Befehl *LES* und indirekte **far**-Aufrufe.

Zugriffe auf Strukturelemente

Der Zugriff auf Strukturelemente kann auch innerhalb von **asm**-Anweisungen auf die übliche Art stattfinden, also über die Angabe von *Variable.Element*. In diesem Fall wird das entsprechende Element wie eine normale Variable behandelt, es können also Werte gespeichert und gelesen werden. Der Elementname kann aber auch direkt verwendet werden (ohne den Variablennamen), wenn er als numerische Konstante behandelt wird. In diesem Fall entspricht die Konstante dem Offset (in Bytes) zur Startadresse der Struktur, die das Element enthält. Sehen Sie sich dazu das folgende Programmfragment an:

```
struct myStruct {
   int a_a;
   int a_b;
   int a_c;
} myA ;

myfunc()

{
   ...
   asm  {mov ax, myA.a_b
         mov bx, [di].a_c
        }
   ...
}
```

In dem Beispiel wird ein Strukturtyp namens *myStruct* mit drei Elementen, *a_a*, *a_b* und *a_c* deklariert. Außerdem wird die Variable *myA* vom Typ *myStruct* deklariert. Die erste **asm**-Anweisung lädt den Wert der Variablen *myA.a_b* in das Register *AX*. Die zweite lädt den Wert an der Adresse $[di] + offset(a_c)$ in das Register *BX* (zu der in *DI* gespeicherten Adresse wird der Offset von *a_c* zum Start von *myStruct* addiert). Diese Anweisungen erzeugen folgenden Code:

```
mov ax, DGROUP : myA+2
mov bx, [di+4]
```

Worin liegt nun der Sinn für diese Vorgehensweise? Wird ein Register (wie z.B. *DI*) mit der Adresse einer Struktur vom Typ *myStruct* geladen, kann der Zugriff auf die Elemente direkt über die Elementnamen erfolgen. Der Elementname kann an jeder Stelle verwendet werden, an der eine numerische Konstante im Operanden einer Assembler-Anweisung zulässig ist.

Die Angabe eines Strukturelements muß mit einem Punkt (.) eingeleitet werden, damit der Compiler erkennt, daß hier nicht ein normales C-Symbol, sondern ein Elementname verwendet wird. Elementnamen werden vom Compiler durch den numerischen Offset des Strukturelements ersetzt (der numerische Offset von *a_c* ist 4). Typ-Information geht dabei allerdings verloren. Elemente können demnach in **asm**-Anweisungen als Compile-Zeit-Konstanten eingesetzt werden.

Hier gibt es aber eine Einschränkung: Werden in einer **asm**-Anweisung zwei Strukturen mit demselben Elementnamen verwendet, muß zur Unterscheidung zwischen dem Punkt und dem Elementnamen der Strukturtyp (in runden Klammern) angegeben werden:

```
asm  mov  bx,[di].(struct tm)tm_hour
```

Sprungbefehle und Labels

In **asm**-Anweisungen können alle bedingten und unbedingten Sprungbefehle und Schleifen verwendet werden. Sie sind aber nur innerhalb einer Funktion gültig. Da in **asm**-Anweisung keine Labels definiert werden können, müssen Sprungbefehle zur Kennzeichnung der Sprungziele **goto**-Labels von C verwenden. Ein Sprung zu einem Label, das zuweit entfernt ist, wird automatisch in einen **far**-Sprung umgewandelt. Direkte **far**-Sprünge sind nicht möglich.

Im folgenden Beispiel erfolgt ein Sprung zum **goto**-Label *a*:

```
int     x()
{
a:                          /* das goto-Label "a" */
   ...
   asm jmp a                /* Sprung zum Label "a" */
   ...
}
```

Auch indirekte Sprünge sind zulässig. Dazu kann der Registername als Operand für den Sprungbefehl verwendet werden.

Interrupt-Funktionen

Der Prozessor 8086 reserviert die ersten 1024 Speicheradressen für eine Tabelle, die aus 256 **far**-Zeigern besteht. Diese Zeiger zeigen normalerweise auf Routinen des Betriebssystems (MSDOS/BIOS) und werden als *Interrupt-Vektoren* bezeichnet.

Der Aufruf einer derartigen Routine geschieht über den folgenden Prozessorbefehl:

```
int Interrupt-Nummer
```

Interrupt-Nummer steht für eine Konstante im Bereich 0h bis FFh. Bei diesem Verfahren werden das Code-Segment (CS), der Befehlszeiger (IP) und die Status-Flags automatisch gespeichert. Anschließend erfolgt ein **far**-Sprung zur Adresse, auf die der entsprechende Interrupt-Vektor zeigt.

Der Aufruf von MSDOS-Routinen geschieht z.B. ausschließlich über Interrupts, wobei hier ein »zentraler« Aufrufbefehl existiert, ohne den praktisch kein Programm auskommt:

```
int 21h
```

Viele der möglichen 256 Interrupt-Vektoren sind nicht belegt. Sie können daher einen eigenen Interrupt-Handler schreiben und einen **far**-Zeiger auf ihn in einem ungenutzten Interrupt-Vektor speichern. Dazu muß eine Funktion vom Typ **interrupt** definiert werden, die folgendermaßen aussieht:

```
void interrupt myhandler (bp, di, si, ds, es, dx,
                          cx, bx, ax, ip, cs, flags, ...);
```

Da alle Register als Parameter übergeben werden, können Sie sie in Ihrem Programm ohne den Einsatz von Pseudovariablen (siehe oben) verwenden und modifizieren. Außerdem können weitere Parameter, die entsprechend definiert sind (*flags*, ...), an die Interrupt-Funktion übergeben werden.

Eine Funktion vom Typ **interrupt** rettet (zusätzlich zu *SI*, *DI* und *BP*) die Register *AX* bis *DX*, *ES* und *DS* automatisch. Sie werden nach der Ausführung der Interrupt-Funktion wiederhergestellt.

Interrupt-Funktionen sind nicht an ein bestimmtes Speichermodell gebunden und können mit Fließkomma-Arithmetik arbeiten – in diesem Fall muß aber dafür gesorgt werden, daß sämtliche Register des 80x87 zwischengespeichert und nach Ablauf der Funktion wiederhergestellt werden.

Eine Interrupt-Funktion kann ihre Parameter verändern. Die entsprechenden Register werden nach dem Ende der Funktion modifiziert. Dies kann sinnvoll sein, wenn die Interrupt-Funktion als Benutzer-Service eingesetzt wird (wie z.B. der DOS-Interrupt 21h). Interrupt-Funktionen enden mit dem Prozessorbefehl *iret*.

Die meisten speicherresidenten Routinen funktionieren, indem sie sich selbst als Interrupt-Funktion installieren. Auf diese Weise fangen diese Routinen bei speziellen oder periodisch auftretenden Ereignissen (wie z.B. bei »Clock-Ticks«, Tastenbetätigungen, etc.) den Aufruf der ursprünglich mit dem Interrupt verbundenen Routine ab und überprüfen, welche Aktion ausgeführt werden soll. Danach geben sie die Steuerung wieder an die ursprüngliche Routine zurück.

Low-Level-Praktiken am Beispiel

Auf den vorhergehenden Seiten wurden einige Beispiele für Low-Level-Praktiken erläutert. Wir wollen uns zum Abschluß noch einige weitere Bespiele ansehen. Beginnen wir mit einer Interrupt-Funktion, die beim Aufruf einen Piepton erzeugt.

Im ersten Schritt wird die Funktion geschrieben:

```
#include <dos.h>

void interrupt mybeep (unsigned bp, unsigned di, unsigned si,
                       unsigned ds, unsigned es, unsigned dx,
                       unsigned cx, unsigned bx, unsigned ax)
{
   int i, j;
   char originalbits, bits;
   unsigned char bcount = ax >> 8;

   /* Lesen des Originalzustands von Port 0x61 */
   bits = originalbits = inportb(0x61);

   for (i = 0; i <= bcount; i++) {

       /* Lautsprecher ausschalten */
       outportb(0x61, bits & 0xfc);
       for (j = 0; j <= 100; j++)
          ;   /* warten */

       /* Lautsprecher an */
       outportb(0x61, bits | 2);
       for (j = 0; j <= 100; j++)
          ;   /* warten */
       }

   /* Wiederherstellung des Originalzustands von Port 0x61 */
   outportb(0x61, originalbits);
}
```

Des weiteren brauchen wir eine Funktion zur Installation der Interrupt-Funktion. Sie bekommt die Adresse der Funktion und die Nummer ihres Interrupt-Vektors (0 bis 255 bzw. 0x00 bis 0xFF) übergeben:

```
void install(void interrupt (*faddr) (), int inum)
{
   setvect(inum, faddr);
}
```

Schließlich müssen wir die *mybeep*-Routine noch testen. Dazu schreiben wir folgende Funktion:

```
void testbeep(unsigned char bcount, int inum)
{
   _AH = bcount;
   geninterrupt(inum);
}
```

Die *main*-Funktion könnten nun so aussehen:

```
main()
{
   char ch;
```

```
    install(mybeep,10);
    testbeep(3,10);
    ch = getch();
}
```

Mit den Funktionen *getvect* und *setvect* könnte der ursprüngliche Interrupt-Vektor gerettet und nach dem Ende des Hauptprogramms (*main*) wiederhergestellt werden.

8. Erstellen einer Windows-Anwendung

In diesem Kapitel erfahren Sie, wie Sie mit Turbo C++ eine Windows-Anwendung oder eine dynamische Linkbibliothek (DLL für »dynamik link library«) erstellen können.

Das Kapitel erläutert jedoch weder die Feinheiten für den Entwurf einer Windows-Anwendung noch erhalten Sie darin eine Anleitung für die Programmierung unter Windows – diese Themen würden den Rahmen des Kapitels und des gesamten Handbuches sprengen.

Ein Programm wird im Prinzip unter Windows genauso compiliert und gelinkt wie unter DOS. Der Compiler generiert als erstes eine .OBJ-Datei, die sich von einer DOS-Compilierung in erster Linie durch den speziellen Windows-Prolog- und Windows-Epilog-Code unterscheidet, der jede Funktion »umschließt«. Prolog- und Epilog-Code hängen davon ab, welche Compileroptionen verwendet werden; die verfügbaren Parameter werden weiter hinten in diesem Kapitel erläutert.

Um für das Speichermodell, unter dem die Compilierung durchgeführt wird, ein Windows-Modul zu erzeugen, werden folgende Schritte durchgeführt:

- Zuerst linkt der Linker die .OBJ-Dateien mit dem entsprechenden Turbo C++ Startup-Code, mit den verschiedenen Bibliotheken und der Moduldefinitionsdatei.
- Danach verbindet der Ressourcenlinker der integrierten Entwicklungsumgebung (IDE) die Ressourcen mit dem Modul.

In Abbildung 8.1 ist dieser Vorgang im Überblick dargestellt.

Im nächsten Abschnitt »Compilieren und Linken in der IDE« wird anhand eines kurzen Beispiels demonstriert, wie ein Windows-Programm in der Turbo C++ IDE compiliert, gelinkt und ausgeführt wird.

Compilieren und Linken in der IDE

Eine vollständige Beschreibung der verfügbaren IDE-Befehle finden Sie in Kapitel 1 »Die integrierte Entwicklungsumgebung« des *Benutzerhandbuches*.

Wir werden eine einfache Anwendung mit dem Namen WHELLO erstellen, die lediglich ein Fenster aufbaut und Text in dieses Fenster ausgibt.

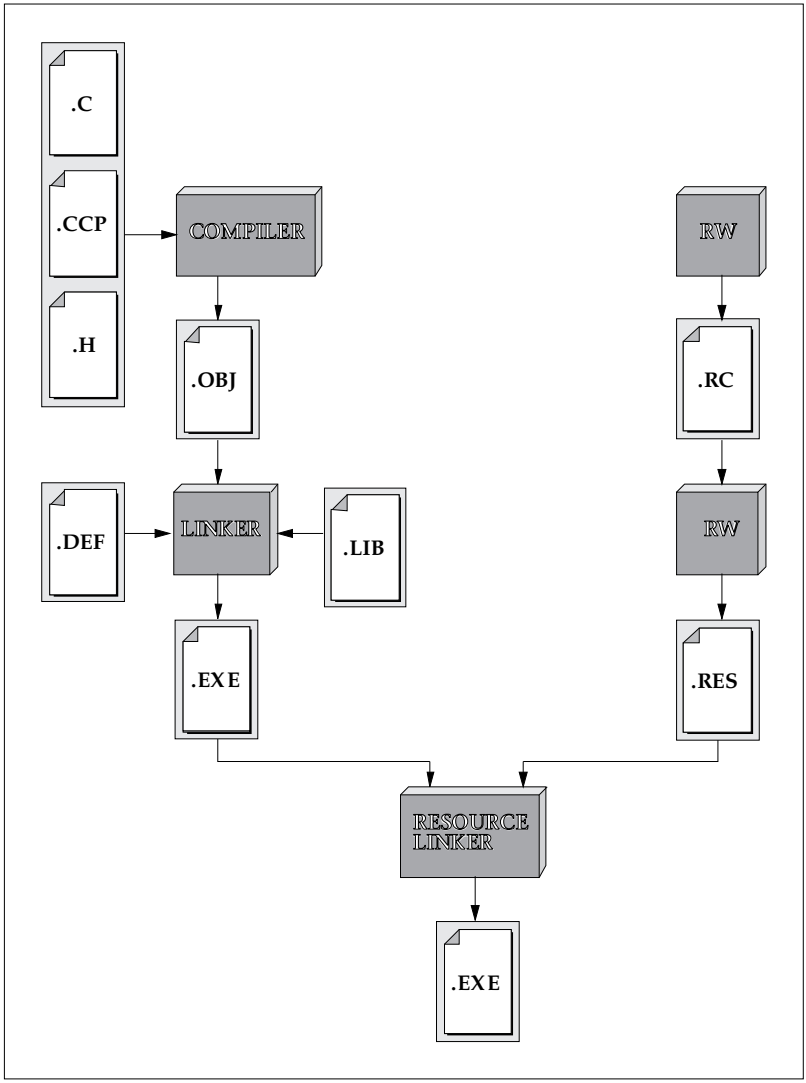


Abbildung 8.1 Ein Windows-Programm compilieren und linken

WHELLO.EXE wird durch Compilieren und Linken der folgenden drei Dateien erzeugt:

- WHELLO.CPP, die C++Quelltextdatei
- WHELLO.RC, die Ressourcendatei
- WHELLO.DEF, die Moduldefinitionsdatei

Die Ressourcen-dateien

Windows-Anwendungen setzen sog. *Ressourcen* (Hilfsmittel) ein. Darunter versteht man Menüs, Dialogfenster, Schriften, verschiedene Mauszeiger, Bitmaps oder benutzerdefinierte Ressourcen. Ihre Definition erfolgt in einer Datei, die Ressourcendatei genannt wird. In unserem Beispiel ist dies WHELLO.RC.

Ressourcendateien mit der Namenserweiterung .RC sind Quelltextdateien, die auch Ressourcen-Script-Dateien heißen. Bevor eine .RC-Datei einer ausführbaren Datei hinzugefügt werden kann, muß sie mit dem Resource Workshop in ein binäres Format gebracht werden. Das Ergebnis der Compilierung ist eine .RES-Datei. So wird durch die Compilierung der Datei WHELLO.RC mit dem Resource Workshop die Datei WHELLO.RES erzeugt. Der Ressourcenlinker der IDE verbindet WHELLO.RES mit WHELLO.EXE.

Die Modul-definitionsdatei

Die Moduldefinitionsdatei WHELLO.DEF informiert den Linker über die Inhalte und die Systemanforderungen einer Windows-Anwendung. Da der Linker auf die in der Datei vorhandenen Informationen auf andere Weise zugreifen kann, benötigt der Turbo C++ Linker keine Moduldefinitionsdateien zur Erstellung einer Windows-Anwendung. Wir haben in unserem Beispiel zu Demonstrationszwecken eine derartige Datei verwendet.

Compilieren und Linken von WHELLO

Führen Sie folgende Arbeitsschritte aus, um aus den drei angesprochenen Dateien eine lauffähige Anwendung zu erstellen:

1. Wählen Sie *Projekt/Projekt öffnen*. Geben Sie im Feld *Dateiname* `WHELLO.PRJ` ein. Drücken Sie CR oder klicken Sie *OK* an, um ein neues Projekt mit dem Namen WHELLO zu öffnen.

2. Wählen Sie *Projekt/Datei einfügen* und geben Sie `whello.*` im Feld *Dateiname* ein. Es wird eine Liste angezeigt, die alle WHELLO-Dateien enthält. Drücken Sie CR oder klicken Sie *OK* an.

3. Fügen Sie die drei Dateien WHELLO.CPP, WHELLO.RES und WHELLO.DEF für die Anwendung ein. Schließen Sie nach dem Einfügen der Dateien das Dialogfenster.

4. Wählen Sie *Optionen/Anwendung*. Das Dialogfenster *Optionen für Anwendung* wird geöffnet. Wählen Sie *Windows EXE* und warten Sie ab, wie sich die Information im oberen Bereich des Dialogfensters ändert. Alle Schalter im unteren Bereich des Dialogfensters markieren und setzen verschiedene weitere Optionen in der IDE.

5. Wählen Sie *Start/Ausführen*, um das Projekt aufzubauen und die vollständige WHELLO.EXE-Anwendung auszuführen.

Mehr müssen Sie nicht tun, um eine Windows-Anwendung mit Turbo C++ zu erstellen und auszuführen. Wenn man den erläuterten Vorgang schematisiert, läßt sich dafür folgende Checkliste zusammenstellen:

1. Erstellen eines Projekts.

2. Erstellen der Ressourcen mit Resource Workshop oder Compilieren der .RC-Dateien zu .RES-Ressourcendateien.

3. Hinzufügen der Quelltext-, Ressourcendateien, der Importbibliotheken (falls erforderlich) und der Moduldefinitionsdatei (falls erforderlich).

4. Festlegen der Compiler- und Linkerumgebung über das Dialogfenster *Optionen für Anwendung* oder mit einer Kombination anderer Einstellungen und Optionen.

5. Aufbauen des Projekts.

6. Ausführen der Anwendung unter Windows.

Verwendung der Projektverwaltung

Die Spezifizierung einer .RES-Datei veranlaßt die Projektverwaltung, die Ressourcen mit dem internen Ressourcenlinker zu der ausführbaren Datei zu binden. Wenn Sie eine .RC-Datei angeben, sucht die Projektverwaltung nach der entsprechenden .RES-Datei. Mit Resource Workshop können Sie aus .RC-Dateien .RES-Dateien erzeugen. Wenn Sie zum Beispiel HELLO.CPP, HELLO.RC und HELLO.DEF in ein Projekt einbauen, führt die Turbo C++Projektverwaltung folgende Arbeitsschritte aus:

- Durch Compilieren von HELLO.CPP mit dem C++Compiler wird HELLO.OBJ angelegt.
- Durch Linken von HELLO.OBJ mit den entsprechenden Bibliotheken (unter Verwendung der Informationen in der Datei HELLO.DEF) wird HELLO.EXE angelegt.
- Durch Verwendung des Ressourcenlinkers, der die in HELLO.RES und HELLO.EXE enthaltenen Ressourcen linkt, wird die endgültige HELLO.EXE- Datei erzeugt.

Mit Resource Workshop compilieren Sie die .RC-Datei zu einer .RES-Datei.

Setzen der Compiler- und Linkeroptionen

Die Einstellungen werden in unserem Beispiel im Dialogfenster *Optionen für Anwendung* vorgenommen. Die Schalter in diesem Dialogfenster steuern oder definieren verschiedene weitere Optionen in anderen Dialogfenstern. Mit Turbo C++ können Sie die Einstellungen, die das Compilieren und Linken des Programms betreffen, auf einfache Weise ändern.

Über die Bedeutung dieser Einstellungen können Sie sich informieren, wenn Sie sich mit den folgenden Dialogfenstern beschäftigen (siehe Kapitel 2 »Menü-Referenz« im *Benutzerhandbuch*):

- In dem Dialogfenster *Optionen für Code-Generierung* wird das zu verwendende Speichermodell eingestellt, festgelegt, ob vorcompilierte Header verwendet werden sollen, etc. Wählen Sie *Optionen/Compiler/Code-Generierung*, um dieses Dialogfenster zu öffnen.

- Im Dialogfenster *Generierung von Entry-|Exitcode* bestimmen Sie die Einstellungen für den Prolog- und Epilog-Code und die Export-Optionen. Wählen Sie *Optionen/Compiler/Entry-|Exitcode* und gehen das angezeigte Dialogfenster durch.

- Im Dialogfenster *Linker-Einstellungen* (*Optionen/Linker/Einstellungen*) bestimmen Sie die Ausgabeart des Link-Ergebnisses, wie Windows-.EXE-Datei oder Windows-DLL und eine weitere Anzahl von Linker-Optionen.

WinMain

Die Funktion *WinMain* ist der Einsprungspunkt für eine Windows-Anwendung und muß immer vorhanden sein.

Folgende Parameter werden an *WinMain* übergeben:

```
int PASCAL WinMain (HANDLE hInstance, HANDLE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
```

Die Typen HANDLE und LPSTR sind in windows.h definiert.

- *hInstance* ist das Instanzen-Handle der Anwendung. Jede Instanz einer Windows-Anwendung hat ein eindeutiges Instanzen-Handle. Dieses wird als Argument für verschiedene Windows-Funktionen verwendet und kann zur Unterscheidung zwischen mehreren Instanzen einer gegebenen Anwendung dienen.

- *hPrevInstance* ist das Handle der in der Anwendung vorausgehenden Instanz. *hPrevInstance* ist NULL, wenn es sich um die erste Instanz handelt.

- *lpCmdLine* ist ein **far**-Zeiger auf einen nullterminierten Kommandozeilenstring. Dieser Wert kann festgelegt werden, wenn die Anwendung vom Programm- Manager oder von *WinExec* aufgerufen wird.

- *nCmdShow* ist eine Integerzahl, die festlegt, wie das Anwendungsfenster angezeigt wird.

Der Rückgabewert von *WinMain* wird von Windows nicht verwendet. Er kann sich aber während des Debuggens als hilfreich erweisen, da Turbo Debugger diesen Wert nach der Beendigung Ihres Programms anzeigen kann.

Prologe und Epiloge

Wenn Sie ein C- oder C++ Programm für Windows compilieren, muß dem Compiler mitgeteilt werden, welcher Prolog und Epilog für die Funktionen jedes Moduls erstellt werden soll. Einstellungen in der IDE und Optionen für den

Kommandozeilencompiler steuern die Erzeugung von Prologen und Epilogen. Diese stellen u. a. sicher, daß bei Callback-Funktionen das richtige Datensegment aktiv ist und daß **near**- und **far**-Stackgrenzen für die unterschiedlichen Stack-Mechanismen von Windows markiert sind.

Prologe und Epiloge sind nicht nur für Windows erforderlich; sie müssen auch für Programm-Code angelegt werden, der für den Einsatz unter DOS generiert wird. Ist das Programm jedoch für Windows gedacht, erzeugt der Compiler einen anderen Prolog und Epilog als für einen DOS-Einsatz.

Prologe und Epiloge werden automatisch vom Compiler erzeugt. Die im Code enthaltenen Anweisungen werden über IDE-Optionen gesteuert.

Die folgende Aufzählung beschreibt die Auswirkungen der verschiedenen Optionen für die Erzeugung von Prolog/Epilog-Code. In der IDE werden diese Optionen durch die Anwahl von *Optionen/Compiler/Entry-|Exitcode* aufgerufen.

Windows: alle Funktionen exportierbar

Diese Option erzeugt ein Objektmodul für ein Windows-Programm, bei dem alle **far**-Funktionen exportierbar sind.

Dies ist die allgemeinste Art eines Windows-Objektmoduls, obgleich nicht notwendigerweise die effizienteste. Der Compiler generiert einen Prolog und Epilog für jede **far**-Funktion, die Sie als exportierbar festlegen. Das bedeutet aber nicht, daß alle **far**-Funktionen wirklich exportiert werden, sondern nur, daß die Möglichkeit dazu besteht. Um eine derartige Funktion zu exportieren, müssen Sie entweder das Schlüsselwort **_export** verwenden oder einen Eintrag für den Funktionsnamen im EXPORTS-Bereich der Moduldefinitionsdatei einfügen.

Im nachfolgenden finden Sie im Abschnitt »Das Schlüsselwort **_export**« eine Beschreibung dieses Schlüsselworts.

Windows: explizite Funktionen exportiert

Diese Option erzeugt ein Objektmodul mit Funktionen, die nur dann exportierbar werden, wenn sie mit dem Schlüsselwort **_export** markiert sind.

Da in einem Modul nicht alle der vorhandenen Funktionen exportiert werden, muß der Compiler nur für die Funktionen, die auch wirklich exportiert werden, die Prologe und Epiloge für exportierbare Funktionen anlegen. Das Schlüsselwort **_export** in einer Funktionsdefinition informiert den Compiler darüber, daß für diese Funktionen die speziellen Prologe und Epiloge erforderlich sind. Alle Funktionen, die nicht mit dem Schlüsselwort **_export** markiert sind, erhalten abgekürzte Prologe und Epiloge. Daraus folgt, daß die .OBJ-Datei kleiner wird und sich die Ausführungsgeschwindigkeit etwas erhöht.

Beachten Sie, daß diese Option *nur* zusammen mit dem Schlüsselwort **_export** funktioniert. Diese Option exportiert nicht die Funktionen, die im EXPORTS-Bereich einer Moduldefinitionsdatei aufgelistet sind. Das heißt mit anderen Worten, daß Sie nicht gleichzeitig diese Option verwenden und die

Namen der exportierten Funktionen im EXPORTS-Bereich aufführen können. Falls Sie es doch tun, erzeugt der Compiler Prolog- und Epilog-Code, der mit exportierten Funktionen nicht kompatibel ist. Dadurch kommt es zu fehlerhaftem Verhalten, wenn diese Funktionen aufgerufen werden.

Windows: intelligente Callbacks

Diese Option erzeugt ein Objektmodul mit Funktionen, die intelligente (engl. smart) Callbacks verwenden.

Diese Form von Prologen und Epilogen gehen von der Voraussetzung aus, daß DS = SS gilt. Das bedeutet mit anderen Worten, daß das Standard-Datensegment dasselbe ist wie das Stacksegment. Dadurch entfällt die Notwendigkeit zur Erzeugung eines speziellen Windows-Codes (des sogenannten *Thunk*), wie er für exportierte Funktionen angelegt wird. Diese Form von Prologen und Epilogen kann die Ausführungsgeschwindigkeit erhöhen, weil Funktionsaufrufe in dem Modul nicht über die Thunks umgeleitet werden müssen.

Zu exportierende Funktionen müssen hier nicht mit dem Schlüsselwort **_export** versehen sein und nicht im EXPORTS-Bereich der Moduldefinitionsdatei stehen, weil der Linker keinen Export-Einsprung für diese Funktionen in der ausführbaren Datei anlegen muß.

Den mit Smart Callbacks compilierten und gelinkten Funktionen muß kein Aufruf von *MakeProcInstance* vorausgehen (*MakeProcInstance* schreibt den Prolog einer Funktion so um, daß sie einen intelligenten Callback verwendet). Ausgehend von der Annahme DS = SS, können Sie diese Option nicht für Module in einer dynamischen Linkbibliothek, sondern nur für Anwendungen verwenden. Außerdem dürfen Sie DS nicht explizit in Ihrem Programm ändern (was unter Windows nie zu empfehlen ist).

Für DLLs gibt es keine intelligenten Callbacks, da bei diesen DS != SS vorausgesetzt wird.

Windows DLL: alle Funktionen exportierbar

Diese Option erzeugt ein dynamisches Linkbibliotheks-Objektmodul, bei dem alle Funktionen exportierbar sind. Dieser Prolog- und Epilog-Code wird für Funktionen einer dynamischen Linkbibliothek verwendet. Er unterstützt auch den Export dieser Funktionen. Dies entspricht der nicht-DLL-Option *Windows: alle Funktionen exportierbar*.

Windows DLL: explizit exportierte Funktionen

Dieser Prolog- und Epilog-Code wird für Funktionen einer dynamischen Linkbibliothek verwendet. Dabei müssen jedoch die Funktionen, die wirklich

exportiert werden sollen, mit dem Schlüsselwort **_export** versehen werden. Dies entspricht der nicht-DLL-Option *Windows: explizite Funktionen exportiert* (siehe oben).

Das Schlüsselwort _export

Das Schlüsselwort **_export** wird in einer Funktionsdefinition eingesetzt, um dem Compiler mitzuteilen, daß eine Funktion als exportierbar compiliert werden soll. Außerdem wird dadurch der Linker angewiesen, die Funktion zu exportieren. **_export** steht in der Funktionsdefinition unmittelbar vor dem Funktionsnamen, z.B.:

```
LONG FAR PASCAL _export MainWindowProc( HWND hWnd,
                                        unsigned iMessage,
                                        WORD wParam, LONG lParam )
```

Hinweis: Exportierbare Funktionen müssen als **FAR** deklariert werden.

_export kann auch in einer C++Klassendefinition verwendet werden. Näheres dazu erfahren Sie weiter hinten in diesem Kapitel im Abschnitt »C++Klassen und Zeiger«.

Zusammenfassung: Prologe, Epiloge und Exporte

Der Export von Funktionen gliedert sich in zwei Phasen. In der ersten Phase muß der Compiler den korrekten Prolog und Epilog für die Funktion erzeugen; wenn das erfolgt ist, wird die Funktion *exportierbar* genannt. In der zweiten Phase muß der Linker einen Eintrag für jede Export-Funktion im Header- Bereich der ausführbaren Datei anlegen. Beide Schritte erfolgen so, daß das richtige Datensegment zur Laufzeit an die Funktion gebunden werden kann.

Wenn eine Funktion mit dem Schlüsselwort **_export** und mit einer der Windows-Compileroptionen markiert ist, wird sie als exportierbare Funktion compiliert und als Export gelinkt.

Ist eine Funktion *nicht* mit dem Schlüsselwort **_export** versehen, führt Turbo C++eine der folgenden Aktionen durch:

- Erfolgt die Compilierung mit den Optionen *Alle Funktionen exportierbar*, wird die Funktion als exportierbare Funktion compiliert. Wenn die Funktion im EXPORTS-Bereich der Moduldefinitionsdatei aufgeführt ist, wird sie als Export gelinkt. Ist sie aber nicht aufgelistet oder wird keine Moduldefinitionsdatei gelinkt, wird die Funktion nicht als Export gelinkt.

- Erfolgt die Compilierung mit den Optionen *Windows: explizite Funktionen exportiert* oder *Windows DLL: explizit exportierte Funktionen*, wird die Funktion *nicht* als exportierbare Funktion compiliert. Ist die Funktion im EXPORTS-Bereich der Moduldefinitionsdatei aufgeführt, wird sie exportiert, wobei es allerdings zu einem Programmfehler kommt, weil der Prolog nicht korrekt ist.

Sie erhalten dann von Windows wahrscheinlich die Fehlermeldung: `Nicht behebbarer Fehler im Anwendungsprogramm .`

In der folgenden Tabelle sind die Auswirkungen der verschiedenen Kombinationsmöglichkeiten von Windows-Compileroptionen und dem Schlüsselwort **_export** zusammengefaßt.

Tabelle 8.1 Compileroptionen und das Schlüsselwort _export

Funktion mit **_export** und **far**?	Ja	Ja	Ja	Ja	Nein	Nein	Nein	Nein
Funktion in EXPORTS?	Ja	Ja	Nein	Nein	Ja	Ja	Nein	Nein
Die *Entry-/Exitcode*-Option ist *	Alle	Expl	Alle	Expl	Alle	Expl	Alle	Expl
Funktion exportierbar?	Ja	Ja	Ja	Ja	Ja	Nein	Ja	Nein
Wird die Funktion exportiert?	Ja	Ja	Ja	Ja	Ja	Ja**	Nein°	Nein

* *Alle* bedeutet *Windows: alle Funktionen exportierbar* oder *Windows DLL: alle Funktionen exportierbar*, *Expl* (*Explizit*) bedeutet *Windows: explizite Funktionen exportiert* oder *Windows DLL: explizit exportierte Funktionen*. Diese Optionen werden im Dialogfenster *Generierung von Entry-|Exitcode* ausgewählt.

** Die Funktion wird exportiert, aber sie funktioniert nicht wie erwartet, da Prolog und Epilog nicht korrekt sind.

° Diese Kombination ist auch wenig sinnvoll. Es ist einfach nicht effizient, alle Funktionen als Export-Funktionen zu definieren, wenn Sie nicht einige davon wirklich exportieren wollen.

Speicher-modelle

Sie können die Speichermodelle *Small*, *Medium*, *Compact* oder *Large* für jede Windows-.EXE-Datei inklusive dynamischer Linkbibliotheken verwenden.

Moduldefinitions-dateien

Eine Moduldefinitionsdatei ist für den Aufbau einer ausführbaren Windows-Datei unter Turbo C++ nicht unbedingt erforderlich.

Wird keine Moduldefinitionsdatei angegeben, gelten folgende Voreinstellungen:

CODE	PRELOAD MOVEABLE DISCARDABLE
DATA	PRELOAD MOVEABLE MULTIPLE (für Anwendungen) oder PRELOAD MOVEABLE SINGLE (für DLLs)
HEAPSIZE	4096
STACKSIZE	5120

Der Linker von Turbo C++ kann – als Ersatz für die EXETYPE-Anweisung – anhand der Einstellungen in der IDE oder der Parameter in der Kommandozeile erkennen, welche Art ausführbarer Datei Sie wünschen.

Sie können anstelle des IMPORTS-Bereichs in der Moduldefinitionsdatei eine Importbibliothek einfügen. Sie können in den Definitionen von Exportfunktionen in Ihrem C- oder C++Programmtext das Schlüsselwort **_export** verwenden und damit den EXPORTS-Bereich ersetzen. Beachten Sie jedoch, daß bei Verwendung von **_export** die Funktion über ihren Namen und nicht über eine Ordinalzahl (was normalerweise effizienter ist) exportiert wird.

Wenn Sie diese voreingestellten Attribute ändern möchten, müssen Sie eine Moduldefinitionsdatei verwenden.

Ein kurzes Beispiel

Im folgenden sehen Sie eine Moduldefinition für das Beispiel WHELLO:

```
NAME            WHELLO
DESCRIPTION     'C++ Windows Hello World'
EXETYPE         WINDOWS
CODE            MOVEABLE
DATA            MOVEABLE MULTIPLE
HEAPSIZE        1024
STACKSIZE       5120
EXPORTS         MainWindowProc
```

Sehen wir uns diese Datei nun Anweisung für Anweisung an:

- NAME legt den Namen für eine Anwendung fest. Wenn Sie eine dynamische Linkbibliothek erstellen möchten, muß anstelle von NAME die Anweisung LIBRARY verwendet werden. Eine Moduldefinition muß eine der beiden Anweisungen enthalten. Der spezifizierte Name muß derselbe Name wie für die .EXE-Datei sein.

- DESCRIPTION ermöglicht die Angabe eines Strings, mit dem Sie Ihre Anwendung oder dynamische Linkbibliothek beschreiben können.

- EXETYPE ist entweder WINDOWS oder OS/2. Die vorliegende Version von Turbo C++ unterstützt nur WINDOWS.

- CODE definiert die Standardattribute für die Code-Segmente. Die Option MOVEABLE bedeutet, daß das Code-Segment zur Laufzeit innerhalb des Speichers verschoben werden kann.

- DATA legt die Standardattribute der Datensegmente fest. MOVEABLE bedeutet, daß das Datensegment zur Laufzeit innerhalb des Speichers verschoben werden kann. Unter Windows können mehrere Instanzen einer Anwendung gleichzeitig ausgeführt werden. Dabei stellt die Option MULTIPLE sicher, daß jede Anwendungs-Instanz ihr eigenes Datensegment hat.

- HEAPSIZE definiert die Größe des lokalen Heap für die Anwendung.

- STACKSIZE bestimmt die Größe des lokalen Stack für die Anwendung. Für die Erstellung einer dynamischen Linkbibliothek kann diese Option nicht verwendet werden.

- EXPORTS führt die Funktionen in der Anwendung WHELLO auf, die von anderen Anwendungen oder von Windows aufgerufen werden. Funktionen, die von anderen Modulen aufgerufen werden, werden Callbacks, Callback-Funktionen oder Export-Funktionen genannt.

- Mit dem Schlüsselwort **_export** kann das Anlegen und Verwalten umfangreicher EXPORTS-Bereiche umgangen werden. Der Linker erkennt mit **_export** markierte Funktionen und trägt sie in eine Exporttabelle für das Modul ein. Bei Verwendung der Option *Intelligente Callbacks* (*Optionen/Compiler/Entry-|Exit-code/Windows: intelligente Callbacks*) müssen Export-Funktionen weder im EXPORTS-Bereich aufgeführt noch mit dem Schlüsselwort **_export** markiert werden. Turbo C++ compiliert die Funktionen so, daß sie als Callback-Funktionen zur Verfügung stehen.

Die vorliegende Anwendung hat keine IMPORTS-Anweisung, weil die von ihr aufgerufenen Funktionen alle aus der Windows-API stammen; diese Funktionen werden aufgrund der automatisch verwendeten Importbibliothek IMPORT.LIB importiert. Ruft eine Anwendung andere externe Funktionen auf, müssen diese im IMPORTS-Bereich aufgeführt oder über eine Importbibliothek eingeführt werden (vgl. den Abschnitt »IMPORTLIB (Importbibliotheken)« weiter hinten in diesem Kapitel).

Die Anwendung hat keine STUB-Anweisung. In dieser Anweisung kann eine DOS-EXE-Datei angegeben werden. Deren Aufgabe ist es zu überprüfen, ob Windows geladen ist, wenn nicht, die Anwendung abzubrechen und eine entsprechende Fehlermeldung auszugeben. Wenn Sie ein anderes als das von Turbo C++ standardmäßig verwendete Programm benutzen wollen, müssen Sie seinen Namen in einer STUB-Anweisung angeben.

Linken für Windows

Normalerweise compiliert Turbo C++ zuerst .OBJ-Dateien mit den entsprechenden Windows-Optionen und linkt diese dann mit dem passenden Windows-Initialisierungs-Code, den passenden Laufzeit- und Arithmetikbibliotheken und einer Moduldefinitionsdatei. In der IDE sorgen Einstellungen in dem Dialogfenster *Linker-Einstellungen* dafür, daß diese Vorgänge automatisch ausgeführt werden.

Linken in der IDE

Mit Hilfe des Dialogfensters *Linker-Einstellungen* in der IDE können Sie Linker-Optionen für eine Windows-Anwendung oder eine dynamische Linkbibliothek setzen. Optionen in der IDE haben Vorrang vor Einstellungen in der Moduldefinitionsdatei. Das hat folgende Konsequenz: Wenn Sie das Schaltfeld *Windows-EXE* anstelle von *Windows-DLL* markieren und die Moduldefinitions-

datei eine LIBRARY- anstelle einer NAME-Anweisung hat, wird die Datei als Anwendung und nicht als dynamische Linkbibliothek gelinkt.

Der Linker verwendet für Anwendungen die Initialisierungsdatei C0Wx.OBJ und für dynamische Linkbibliotheken die Initialisierungsdatei C0Dx.OBJ. Dabei hängt *x* von dem Speichermodell ab, das im Dialogfenster *Optionen für Code-Generierung* festgelegt wurde. Für beide Windows-Optionen benützt der Linker die aktuellen Projekt- .OBJ-Dateien und die Bibliotheken IMPORT.LIB, MATHWx.LIB und CWx.LIB. Turbo C++ ermöglicht es Ihnen, die voreingestellten Speichermodell-Einstellungen zu überschreiben.

Dynamische Linkbibliotheken (DLL)

Eine dynamische Linkbibliothek (DLL) ist eine Bibliothek mit Funktionen, die von Ihrer Windows-Anwendung zur Ausführung einer Aufgabe aufgerufen werden können. Haben Sie bereits eine Windows-Anwendung erstellt, dann haben Sie auch schon dynamische Linkbibliotheken benutzt. Die Dateien KERNEL.EXE, USER.EXE und GDI.EXE sind in Wirklichkeit .DLL-Dateien, also keine Anwendungen (wie die Namenserweiterung .EXE impliziert). Die API-Funktionen, die Sie aus diesen Modulen aufrufen, werden zur Laufzeit gelinkt (dynamisches Linken), und nicht beim Linken der Anwendung (statisches Linken).

Compilieren und Linken einer dynamischen Linkbibliothek in der IDE

Führen Sie folgende Arbeitsschritte aus, um eine Linkbibliothek in der IDE zu compilieren und zu linken:

1. Erstellen Sie die Quelltextdateien der dynamischen Linkbibliothek. Optional können Sie auch die Ressourcen- und Moduldefinitionsdatei anlegen.

2. Beginnen Sie durch die Auswahl von *Projekt/Projekt öffnen* ein neues Projekt.

3. Wählen Sie *Projekt/Datei einfügen*, und tragen Sie die Quelltext- und Ressourcendateien für die dynamische Linkbibliothek ein.

4. Wenn Sie eine Moduldefinitionsdatei für die Linkbibliothek angelegt haben, fügen Sie diese dem Projekt hinzu. (Turbo C++ kann auch ohne Moduldefinitionsdatei linken. Wenn Sie ohne Moduldefinitionsdatei linken, muß jede zu exportierende Funktion in der dynamischen Linkbibliothek mit dem Schlüsselwort **_export** als exportierbar gekennzeichnet sein. Das Schlüsselwort muß dabei unmittelbar vor dem Funktionsnamen stehen. Wählen Sie außerdem *Optionen/Compiler/Entry-|Exitcode/Windows DLL: explizit exportierte Funktionen*.)

5. Wählen Sie *Optionen/Anwendung/Windows DLL.*

6. Wählen Sie *Compiler/Alle Projekt-Dateien compilieren*.

Importbibliotheken

Wenn ein Windows-Anwendungmodul oder eine dynamische Linkbibliothek Funktionen aus einer dynamischen Linkbibliothek verwendet, haben Sie zwei Möglichkeiten, dies dem Linker mitzuteilen:

- Sie können einen IMPORTS-Bereich in die Moduldefinitionsdatei einfügen und alle Funktionen aus dynamischen Linkbibliotheken auflisten, die von dem Modul verwendet werden.
- Sie können die Importbibliothek für die dynamischen Linkbibliotheken angeben, wenn Sie das Modul linken. Im folgenden Abschnitt erfahren Sie, wie Sie eigene Importbibliotheken anlegen können.

Anlegen einer Importbibliothek

IIMPLIBW, die Import-Bibliotheksverwaltung für Windows, legt eine Importbibliothek an, die in einer Windows-Anwendung den IMPORTS-Bereich einer Moduldefinitionsdatei ganz oder teilweise ersetzt.

Eine Importbibliothek listet einige oder alle exportierten Funktionen für eine oder mehrere dynamische Linkbibliotheken auf. IMPLIBW erzeugt eine Importbibliothek direkt aus den dynamischen Linkbibliotheken oder den Moduldefinitionsdateien für dynamische Linkbibliotheken (oder aus einer Kombination von beiden).

Auswählen einer Importbibliothek

Klicken Sie das Symbol IMPLIBW zweimal an, wenn Sie eine Importbibliothek anlegen möchten. Es wird ein leerer Bildschirm angezeigt. Wählen Sie *Datei/Öffnen* an. Es wird ein Dialogfenster eingeblendet, in dem alle dynamischen Linkbibliotheken des aktuellen Verzeichnisses aufgeführt sind. Sie können wie üblich Verzeichnisse wechseln, indem Sie die Verzeichnisnamen in der Liste zweimal anklicken.

Importbibliothek aus einer dynamischen Linkbibliothek anlegen

Wenn Sie eine dynamische Linkbibliothek auswählen und *Öffnen* (oder zweimal den Dateinamen der Linkbibliothek) anklicken, legt IMPLIBW eine Importbibliothek an, die denselben Basisnamen wie die ausgewählte Linkbibliothek und die Namenserweiterung .LIB hat.

Importbibliothek aus einer Moduldefinitionsdatei anlegen

Wenn Sie für eine dynamische Linkbibliothek eine Moduldefinitionsdatei (.DEF) einsetzen, können Sie diese anstelle der Linkbibliothek für das Anlegen einer Importbibliothek verwenden.

Geben Sie den Namen einer .DEF-Datei in das Editierfeld ein und klicken Sie *Öffnen* an. IMPLIBW legt eine Importbiblothek an, die denselben Basisnamen wie die ausgewählte .DEF-Datei und die Namenserweiterung .LIB hat.

Anlegen der Importbibliothek

Wenn IMPLIBW bei der Überprüfung der dynamischen Linkbibliothek oder der Moduldefinitionsdatei und der angelegten Importbibliothek keine Fehler findet, erhalten Sie eine entsprechende Meldung. Sie sind nun im Besitz einer neuen Inportbibliothek, die Sie in ein Projekt einbinden können.

Anlegen von dynamischen Linkbibliotheken

Im folgenden Abschnitt gehen wir auf einige Besonderheiten beim Schreiben einer dynamischen Linkbibliothek ein.

LibMain und WEP

Die Funktion *LibMain* ist der Haupteinsprungspunkt für eine dynamische Linkbibliothek.

Windows ruft *LibMain* einmal auf, wenn die Bibkliothek zum ersten Mal geladen wird. *LibMain* führt die Initialisierung für die dynamische Linkbibliothek aus. Diese Initialisierung hängt vollkommen von der Funktion der speziellen Linkbibliothek ab, umfaßt aber immer folgende Aufgaben:

- Freigeben des Datensegments mit *UnlockData*, wenn dieses als MOVEABLE deklariert wurde.
- Setzen von globalen Variablen für die dynamische Linkbibliothek, falls welche verwendet werden.

Hinweis: Der Linkbibliothek-Startup-Code C0Dx.OBJ initialisiert den lokalen Heap automatisch; Sie müssen dafür keinen Code in *LibMain* eingeben.

Folgende Argumente werden an *LibMain* übergeben:

```
int FAR PASCAL LibMain (HANDLE hInstance, WORD wDataSeg,
                        WORD cbHeapSize, LPSTR lpCmdLine)
```

HANDLE, WORD und LPSTR sind in windows.h definiert.

- *hInstance* ist das Instanzen-Handle der dynamischen Linkbibliothek.
- *wDataSeg* ist der Wert des Datensegment-Registers (DS).
- *cbHeapSize* ist die Größe des lokalen Heap, der in der Moduldefinitionsdatei für die dynamische Linkbibliothek angegeben wird.
- *lpCmdLine* ist ein **far**-Zeiger auf die Kommandozeile, die beim Laden der dynamischen Linkbibliothek eingetragen wurde. Diese Zeile ist in dem meisten Fällen leer, da dynamische Linkbibliotheken typischerweise automatisch ohne Parameter geladen werden. Soll eine dynamische Linkbibliothek explizit geladen werden, dann kann ihr jedoch eine Kommandozeile übergeben werden.

Der Rückgabewert von *LibMain* ist entweder 1 (erfolgreiche Initialisierung) oder 0 (keine Initialisierung). Bei dem Ergebnis 0 entfernt Windows die dynamische Linkbibliothek aus dem Speicher.

Der Austrittspunkt einer dynamischen Linkbibliothek ist die Funktion *WEP*

(»Windows Exit Procedure«). Diese Funktion muß nicht unbedingt in einer Linkbibliothek vorhanden sein (weil die Turbo C++ Laufzeitbibliotheken über eine Standardfunktion verfügen). Sie kann aber angegeben werden, um die dynamische Linkbibliothek vor dem Entfernen aus dem Speicher »aufzuräumen«. Windows ruft die *WEP*-Prozedur direkt vor dem Entfernen der dynamischen Linkbibliothek auf. Bei Turbo C++ muß *WEP* nicht exportiert werden. Turbo C++ definiert seine eigene *WEP*, die wiederum Ihre *WEP* aufruft, und führt dann die System-Aufräumarbeiten durch.

Der Prototyp der Funktion *WEP* hat folgendes Aussehen:

```
int FAR PASCAL WEP (int nParameter)
```

- *nParameter* ist entweder WEP_SYSTEMEXIT oder WEP_FREE_DLL. Ersteres bedeutet, daß das gesamte Windows heruntergefahren wird, zweiteres bedingt nur das Entfernen der dynamischen Linkbibliothek. *WEP* sollte 1 zurückliefern, um die erfolgreiche Durchführung anzuzeigen. Derzeit verwendet Windows diesen Rückgabewert allerdings nicht.

Zeiger und Speicher

Funktionen in einer dynamischen Linkbibliothek werden nicht direkt in eine Windows-Anwendung gelinkt, sondern zur Laufzeit aufgerufen. Das bedeutet, daß Aufrufe von dynamischen Linkbibliotheksfunktionen **far**-Aufrufe sind, weil die Linkbibliothek ein anderes Code-Segment als die Anwendung hat. Die von den aufgerufenen Linkbibliotheksfunktionen verwendeten Daten müssen ebenfalls vom Typ **far** sein.

Nehmen wir z.B. an, daß Sie eine Windows-Anwendung mit dem Namen APP1, eine dynamische Linkbibliothek, definiert von LSOURCE1.C, und eine Header-Datei für diese dynamische Linkbibliothek mit dem Namen lsource1.h haben. Die Funktion *f1*, die auf einem String operiert, wird von der Anwendung aufgerufen.

Wenn die Funktion unabhängig von dem bei der Compilierung der dynamischen Linkbibliothek verwendeten Speichermodell korrekt arbeiten soll, müssen Sie die Funktion und ihre Daten als **far** definieren. In der Header-Datei würde der Funktionsprototyp folgendermaßen aussehen:

```
extern int _export FAR f(char FAR *dstring);
```

In der dynamischen Linkbibliothek hat die Implementation der Funktion folgendes Format:

```
int FAR f1(char far *dstring)
{
    ...
}
```

Damit die Anwendung die Funktion benützten kann, müßte sie außerdem als exportierbar compiliert und dann exportiert werden. Dazu können Sie entweder die dynamische Linkbibliothek so compilieren, daß alle Funktionen exportierbar sind und *f1* im EXPORTS-Bereich der Moduldefinitionsdatei auflisten, oder die Funktion mit dem Schlüsselwort **_export** markieren:

```
int FAR _export f1(char far *dstring)
{
    ...
}
```

Wenn Sie die dynamische Linkbibliothek unter dem Speichermodell *Large* (**far**-Daten, **far**-Code) compilieren, müssen die Funktion und ihre Daten in der dynamischen Linkbibliothek nicht explizit als **far** definiert werden. Der Prototyp in der Header-Datei hätte immer noch folgendes Aussehen:

```
extern int FAR f(char FAR *dstring);
```

denn der Prototyp müßte auch für ein Modul korrekt sein, das mit einem kleineren Speichermodell compiliert wird. In der dynamischen Linkbibliothek kann die Funktion jedoch folgendermaßen definiert werden:

```
int _export f1(char *dstring)
{
    ...
}
```

Hinweis: Bevor eine Anwendung *f1* verwenden kann, muß *f1* in die Anwendung importiert werden. Dies erfolgt entweder durch Auflisten von *f1* im IMPORTS-Bereich der Moduldefinitionsdatei oder Linken einer Importbibliothek für die dynamische Linkbibliothek.

Statische Daten in dynamischen Linkbibliotheken

Über die in den dynamischen Linkbibliotheken enthaltenen Funktionen haben alle Anwendungen, die die Linkbibliothek verwenden, Zugriff auf deren globale Daten. Eine bestimmte Funktion verwendet immer dieselben Daten, gleichgültig, welche Anwendung sie aufruft. Wenn Sie nur einer einzigen Anwendung den Zugriff auf die globalen Daten einer Linkbibliothek erlauben wollen, müssen Sie den Schutzmechanismus selbst schreiben. Die dynamische Linkbibliothek selbst hat keinen Mechanismus, der die globalen Daten nur einer einzigen Anwendung zugänglich macht. Wenn Sie Daten benötigen, die für den Aufrufer einer dynamischen Linkbibliothek privat sein sollen, müssen Sie die Daten dynamisch zuweisen und den Zugriff auf diese Daten manuell verwalten. Statische Daten in einer dynamischen Linkbibliothek sind für alle Aufrufer dieser Linkbibliothek global.

C++ Klassen und Zeiger

Eine C++ Klasse, die nur innerhalb einer dynamischen Linkbibliothek verwendet wird, muß nicht als **far** definiert sein. Soll die Klasse aber auch von einer anderen Linkbibliothek oder einer Windows-Anwendung benützt werden, müssen besondere Maßnahmen ergriffen werden. Alle Datenelemente und Elementfunktionen einer gemeinsam genutzten Klasse müssen als **far** definiert sein. Zu diesem Zweck definieren Sie die Klassenelemente als **far** oder compilieren die dynamische Linkbibliothek mit dem Speichermodell *Large*. Außerdem müssen die Klassen exportiert werden.

Dies kann auf zwei Arten erreicht werden:

- Entweder Sie tragen die Namen aller Klassenelemente in den EXPORTS-Bereich der Moduldefinitionsdatei ein und compilieren die dynamische Linkbibliothek mit der Option *Optionen/Compiler/Entry-|Exitcode/Windows DLL: alle Funktionen exportierbar*), die alle Funktionen als exportierbare Funktionen definiert.

- Oder Sie markieren die gesamte Klasse mit dem Schlüsselwort **_export** und compilieren die dynamische Linkbibliothek mit der Option *Optionen/Compiler/Entry-|Exitcode/Windows DLL: explizit exportierte Funktionen*, die nur bestimmte, explizit angegebene Funktionen als exportierbar definiert.

C++-Klassen verwenden Zeiger auf Tabellen virtueller Funktionen und beinhalten einen versteckten **this**-Zeiger, wobei beide Zeiger als **far** definiert sein müssen. Dies kann auf zweierlei Arten gewährleistet werden.

- Der einfachste Weg besteht darin, die Linkbibliotheksmodule und die die dynamische Linkbibliothek verwendende Anwendung mit der Option *Virtuelle Tabellen als Far* zu compilieren (*Optionen/Compiler/C++ Optionen*). Dadurch werden alle Zeiger auf Tabellen virtueller Funktionen und **this**-Parameter zu 32-Bit-Zeigern. Der Vorteil dieses Ansatzes liegt darin, daß keine Quelltext-Änderungen durchgeführt werden müssen. Der Nachteil besteht darin, daß Sie für alle Klassen, gemeinsam genutzt oder nicht, den Overhead von 32-Bit-Zeigern in Kauf nehmen und mit einem Verlust an Effizienz bezahlen müssen.

- Ein effizienteres Vorgehen ist die Deklaration von gemeinsam genutzten Klassen als **huge**. Damit erhält der Compiler die Information, nur vollständige 32-Bit-Zeiger für diese Klassen zu verwenden.

Beachten Sie, daß eine **huge**-Klasse nur von anderen **huge**-Klassen abgeleitet werden kann und umgekehrt. Im folgenden finden Sie ein Beispiel für die Deklaration einer **huge**-Klasse:

```
class  huge  DLLclass
{
     ... etc ...
};
```

Die Nicht-Inline-Funktionen und die statischen Datenelemente einer Klasse, die in einer dynamischen Linkbibliothek als von einer anderen Anwendung verwendbar definiert ist, müssen verfügbar gemacht werden, indem sie exportierte Namen erhalten. Dies erfolgt durch das Einfügen ihrer öffentlichen Namen in den EXPORTS-Bereich der Moduldefinitionsdatei der Linkbibliothek, was natürlich Zeit kostet und lästig ist.

Es gibt eine einfachere Alternative: Deklarieren Sie die Klassen als **_export**-Klassen. Sobald eine Klasse als **_export** deklariert ist, behandelt Turbo C++ sie als **huge** (mit 32-Bit-Zeigern) und exportiert automatisch alle Nicht-Inline-Elementfunktionen und statischen Datenelemente. Wenn Sie eine Klasse jedoch als **_export** deklarieren, kann diese nicht gleichzeitig als **far** oder **huge** deklariert werden (**_export** impliziert **huge**, das wiederum **far** impliziert).

Wenn Sie die Klasse in einer Include-Datei deklarieren, die sowohl von den Quelltextdateien der dynamische Linkbibliothek als auch von den Quelltextdateien der die Linkbibliothek verwendenden Anwendung eingebunden wird,

sollte die Klasse für das Compilieren einer dynamischen Linkbibliothek als **_export** und für das Compileren einer Anwendung lediglich als **huge** deklariert werden. Dazu haben Sie die Möglichkeit, das Makro __DLL__ einzusetzen, das der Compiler beim Erstellen einer dynamischen Linkbibliothek definiert.

So können Sie beispielsweise den folgenden Code in einer Include-Datei verwenden, die eine gemeinsam genutzte Klasse definiert:

```
#ifdef          __DLL__
#define         EXPORT          _export
#else
#define         EXPORT          huge
#endif

class           EXPORT          DLLclass
{
            ... etc ...
};
```

Beachten Sie, daß der Compiler die Information, daß ein Klassenelement ein Element einer **huge**-Klasse ist, in dem Namen verschlüsselt. Dies gewährleistet, daß beim Linken keine Fehler auftreten, wenn ein Programm mit **huge**- und nicht-**huge**-Klassen gelinkt wird.

9. Die ANSI-C-Implementation bei Turbo C++

Einige Aspekte des ANSI-Standards für die Sprache C sind nicht exakt definiert. Bei jeder Implementation eines ANSI-C-konformen Compilers hat man daher gewisse Freiheiten. Dieses Kapitel erläutert, wie sich Borland in diesen Fällen entschieden hat, den ANSI-C-Standard zu implementieren. Die Gliederungspunkte beziehen sich auf den bisher letzten Entwurf des ANSI-Komitees vom Februar 1990. Die Ausführungen in diesem Kapitel beziehen sich nur auf C, nicht auf C++.

2.1.1.3 Wie soll eine Diagnose identifiziert werden?

Wenn der Compiler mit der korrekten Kombination von gesetzten Optionen arbeitet, ist jede Meldung, die mit den Worten *Fatal*, *Error* oder *Warning* beginnt, eine Diagnose im Sinne des ANSI – Standards. Die Optionen, die das sicherstellen, sind in der folgenden Tabelle aufgeführt. Die Optionen sind als Kürzel aufgelistet. So bezieht sich erste Option auf das Menü *Optionen/Compiler/Quellcode*, wobei Sie im Dialogfenster *Optionen für Quellcode ANSI* als Schlüsselwort gewählt haben.

Tabelle 9.1 Identifizieren der Diagnosen in Turbo C++

Option	Aktion
O/C/Q/A	Nur ANSI Schlüsselwörter.
O/C/Q/V	Keine geschachtelten Kommentare.
O/C/A/C-	Nur C-Aufrufkonventionen.
O/C/Q/B	Wenigstens 32 signifikante Zeichen in Bezeichnern.
O/C/M/A	Alle Warnungen bis auf die folgenden ausschalten.
O/C/M/V/I	Warnung aktivieren über ungeeignete Initialisierung. Warnung aktivieren über Deklarationen ohne Typ oder Speicherklasse.
O/C/M/P/C	Warnung aktivieren über nicht portierbare Zeigervergleiche.
O/C/M/V/R	Warnung aktivieren über nicht-identische Wiederholungen von Definitionen.
O/C/M/V/S	Warnung aktivieren über zweifelhafte Zeigerumwandlung.
O/C/M/P/N	Warnung aktivieren über nicht portierbare Zeigerumwandlung.
O/C/M/V/V	Warnung aktivieren über **void**-Funktionen, die einen Wert zurückliefern.
O/C/M/V/H	Warnung aktivieren über hexadezimale Werte, die mehr als drei Ziffern enthalten.

Option	Aktion
O/C/M/P/M	Warnung aktivieren über das Mischen von Zeigern auf **signed** und **unsigned char**.
O/C/M/V/U	Warnung aktivieren über undefinierte Strukturen.
O/C/M/V/X	Warnung aktivieren über Variablen, die statisch und extern deklariert sind.

Wählen Sie *Optionen/Compiler/Code-Generierung* und legen Sie Sie *Nie* für *SS gleich DS* fest. Ändern Sie außerdem nicht die voreingestellten Werte im Dialogfenster *Segmentnamen* im Menü *Optionen/Compiler/Namen*; jedes Textfeld sollte ein Sternchen (*) beinhalten.

Alle anderen Optionen, die hier nicht erwähnt worden sind, können beliebig verwendet werden.

2.1.2.2.1 Semantik der Argumente von *main*.

Der Wert von *argv*[*0*] ist ein Zeiger auf ein Null-Byte, wenn das Programm unter früheren DOS-Versionen als 3.0 läuft. Ab Version 3.0 zeigt *argv*[*0*] auf den Programmnamen.

Die anderen *argv*-Strings zeigen auf jede Komponente der DOS-Kommandozeilenargumente. Whitespace zwischen den Argumenten wird entfernt, und jede Folge von kontinuierlichen Zeichen (die nicht Whitespace sind) wird als einzelnes Argument behandelt.

Strings in Anführungszeichen werden korrekt behandelt, d.h. sie werden auch mit Leerzeichen akzeptiert.

2.1.2.3 Was konstituiert ein interaktives Gerät?

Alles, was so aussieht wie die Konsole.

2.2.1 Vergleichssequenz für den Zeichensatz des Programmcodes.

Die Vergleichssequenz des Zeichensatzes für ausführbaren Programmcode verwendet den vorzeichenbehafteten Wert des entsprechenden ASCII-Zeichens.

2.2.1 Zugelassener Zeichensatz für ausführbaren Programmcode und Quelltext.

In beiden Fällen der erweiterte ASCII-Zeichensatz des IBM PC. Jedes Zeichen außer ^Z (Control-Z) kann in Literalen, Zeichenkonstanten oder Kommentaren erscheinen.

2.2.1.2 Multibyte-Zeichen (Zeichen, die mehrere Bytes belegen).

Solche Zeichen werden nicht unterstützt.

2.2.2 Ausgaberichtung.

Gedruckt und angezeigt wird von links nach rechts.

2.2.4.2 Zahl der Bits eines Zeichens im Zeichensatz des Programmcodes.
8 Bits pro Zeichen.

3.1.2 Zahl der signifikanten Anfangszeichen eines Bezeichners.
Die ersten 32 Zeichen, wobei diese Zahl mit der Option *O/C/Q/B* verändert werden kann. Interne und externe Bezeichner haben dieselbe Anzahl signifikanter Stellen. (In C++ ist die Zahl der signifikanten Zeichen unbegrenzt.)

3.1.2 Wird zwischen Groß- und Kleinbuchstaben in externen Bezeichnern unterschieden?
Im Normalfall wird der Compiler den Linker dazu zwingen. Man kann die Unterscheidung mit der Option *Groß-/Kleinschreibung* (*O/L/E/L*) unterdrücken.

3.1.2.5 Darstellung und Wertebereiche der verschiedenen Integer-Typen

Datentyp	Größe (Bits)	Minimum	Maximum
signed char	8	-128	127
unsigned char	8	0	255
signed short	16	-32768	32767
unsigned short	16	0	65535
signed int	16	-32768	32767
unsigned int	16	0	65535
signed long	32	-2147483648	2147483647
unsigned long	32	0	4294967295

Alle **char**-Typen verwenden für das Speichern ein 8-Bit-Byte. Alle **short**- und **int**-Typen verwenden 2 Bytes. Alle **long**-Typen verwenden 4 Bytes.

Falls die wortweise Ausrichtung von Speicheradressen gewünscht wird (*O/C/G/G*), werden alle Integertypen, die nicht den Datentyp **char** haben, an geraden Adressen (=1 Wort) ausgerichtet. **char**-Typen werden nie ausgerichtet.

3.1.2.5 Darstellung und Wertebereiche der verschiedenen Gleitkommazahlen.
Das IEEE-Format für Gleitkommazahlen des Intel 8087 wird bei allen Gleitkommazahl-Typen von Turbo C++ unterstützt. Der **float**-Typ benutzt das 32-Bit IEEE-Format (real). Der **double**-Typ verwendet das 64-Bit IEEE-Format (real). Der Datentyp **long double** benutzt das 80-Bit IEEE-Format (extended real).

3.1.3.4 Abbildung des Quelltext-Zeichensatzes auf den des ausführbaren Programmcodes.
Jedes Zeichen in Stringliteralen und Zeichenkonstanten wird unverändert in das ausführbare Programm übernommen. Die Zeichensätze von Quelltext und ausführbaren Programmcode sind identisch.

3.1.3.4 Wert einer Integer-Zeichenkonstanten, die ein Zeichen oder eine Escape-Sequenz enthält, die nicht im grundlegenden Zeichensatz für den ausführbaren Programmcode oder im erweiterten Zeichensatz für *»wide character«-Konstanten* enthalten sind.

»wide *charakter*«-Zeichen werden nicht unterstützt. Sie werden als normale Zeichen behandelt. Alle gültigen Escape-Sequenzen bestehen aus einem oder zwei aufeinanderfolgenden Zeichen. Werden hexadezimale oder oktale Sequenzen verwendet, die den Gültigkeitsbereich eines Zeichens überschreiten, gibt der Compiler eine entsprechende Meldung aus.

3.1.3.4 Aktueller *locale*, der verwendet wird, um Multibyte-Zeichen für eine *»wide character«-Konstante* in die korrespondierenden *wide character*-Zeichen umzuwandeln.

»wide character«-Konstanten werden erkannt, aber wie normale Zeichenkonstanten behandelt. In diesem Sinn ist der *locale* der »C«-*locale*.

3.1.3.4 Wert einer Integer-Konstanten, die mehr als ein Zeichen enthält oder einer *»wide character«-Konstante*, die mehr als ein Multibyte-Zeichen enthält.

Zeichenkonstanten können ein oder zwei Zeichen enthalten. Bei Verwendung von zwei Zeichen werden diese als **int** in 2 Bytes gespeichert, das erste im niederwertigen, das zweite im höherwertigen.

3.2.1.2 Resultat der Umwandlung eines Integer in einen kürzeren *signed*-Integer oder das Resultat der Umwandlung eines *unsigned*-Integer in einen *signed*-Integer gleicher Länge, wenn der Wert nicht dargestellt werden kann.

Diese Umwandlungen werden durch einfaches Abschneiden der höheren Bits ausgeführt. **signed**-Integer werden als Zweierkomplemente gespeichert, auch die resultierende Zahl wird als ein solcher Wert interpretiert. Wenn das höherwertige Bit des kleineren Integer nicht 0 ist, wird es als negativer, ansonsten als positiver Wert interpretiert.

3.2.1.3 Richtung, in der gerundet wird, wenn ein Integer in eine Gleitkommazahl konvertiert wird, die nicht exakt denselben Wert darstellen kann.

Der Integer-Wert wird zum nächsten darstellbaren Wert hin gerundet. Beispielsweise wird der **long**-Wert 2^{31} -1 in den **float**-Wert 2^{31} konvertiert. Zweifelsfälle (wenn der darzustellende Wert genau in der Mitte liegt) werden gemäß den Regeln des IEEE-Standards behandelt.

3.2.1.4 Richtung, in der gerundet wird, wenn eine Gleitkommazahl in eine kürzere Gleitkommazahl konvertiert wird, die nicht exakt den selben Wert darstellen kann.

Der Wert wird zum nächsten darstellbaren Wert hin gerundet. Zweifelsfälle (wenn der darzustellende Wert genau in der Mitte liegt) werden gemäß den Regeln des IEEE-Standards behandelt.

3.3 Resultat von Bitmanipulationen bei *signed*-Integer.
Bit-Operatoren werden auf **signed**-Integer angewendet, als ob sie die korrespondierenden vorzeichenlosen Datentypen wären. Das Bit für das Vorzeichen wird wie ein Datenbit behandelt. Das Resultat wird dann als normaler **signed**-Integer in Zweierkomplement-Darstellung interpretiert.

3.3.2.3 Was passiert, wenn auf ein Element einer Variante (union) über ein Element eines anderen Datentyps zugegriffen wird?
Der Zugriff ist erlaubt und wird einfach die dort gespeicherten Bits ansprechen. Man benötigt ein genaues Verständnis der Bit-Entschlüsselung von Gleitkommazahlen, um kontrolliert auf ein Gleitkommaelement über ein Element eines anderen Typs zuzugreifen. Wenn das gespeicherte Element kürzer ist als das Element, das für den Zugriff auf diesen Wert verwendet wird, dann behalten die zusätzlichen Bits den Wert, den sie vor dem Speichern des kurzen Elements hatten.

3.3.3.4 Datentyp eines Integer-Wertes, der benötigt wird, um die maximale Größe eines Array zu speichern.
Für ein normales Array wird hierzu der Datentyp **unsigned int**, für ein *huge*-Array der Datentyp **signed long** benötigt.

3.3.4 Ergebnis einer Typ-Umwandlung eines Zeigers in einen Integer-Wert und umgekehrt.
Bei Typ-Umwandlungen zwischen Integer-Objekten und Zeigern derselben Größe wird kein Bit verändert. Bei Umwandlungen von längeren zu kürzeren Typen werden die höherwertigen Bits abgeschnitten. Bei einer Umwandlung eines kürzeren Integer in einen längeren Zeiger wird der Integer zunächst auf einen Integer erweitert, der dieselbe Größe hat wie der Zeiger. Das heißt, **signed**-Integer werden um das Vorzeichen erweitert, um die neuen Bytes auszufüllen. In ähnlicher Weise werden kleinere Zeiger-Typen in größere Integer-Typen umgewandelt, indem zunächst der Zeiger auf eine Länge erweitert wird, die dem Integer-Typen entspricht.

3.3.5 Vorzeichen des Rests einer Integer-Division.
Nur, wenn einer der Operanden negativ ist, bekommt der Divisionsrest ein negatives Vorzeichen. Ist keiner oder sind beide Operanden negativ, ergibt sich ein positives Vorzeichen.

3.3.6 Typ eines Integers, das benötigt wird, um die Differenz von zwei Zeigern zu speichern, die auf Elemente desselben Arrays zeigen (*ptrdiff_t*)
Der Datentyp ist **signed int**, wenn die Zeiger **near** sind. Falls diese **far** oder **huge** sind, ist er **signed long**. Der Datentyp von *ptrdiff_t* hängt vom verwendeten Speichermodell ab. Bei den kleinen Speichermodellen ist es der Datentyp **int**, bei den großen **long**.

3.3.7 Resultat einer Rechtsschiebe-Operation eines *signed* Integers mit negativem Vorzeichen.

Ein Wert mit negativem Vorzeichen wird beim Rechtsschieben mit dem Vorzeichen erweitert.

3.5.1 Umfang, in dem Objekte in Registern unter Verwendung der *register*-Deklaration aktuell gespeichert werden können.

Objekte, die als 2 Byte große Integer oder Zeiger-Typen deklariert wurden (**short**-, **int**-, **near**-Zeiger und **unsigned int**), können in Registern gespeichert werden. Der Compiler wird jedes kleinere **auto**-Objekt automatisch in Registern ablegen, jedoch Objekte, die explizit als **register** deklariert wurden, vorrangig behandeln. Mindestens zwei und höchstens sechs Register sind verfügbar. Die Zahl der tatsächlich genutzten Register hängt davon ab, wieviele Register aktuell von temporären Werten der Funktion benötigt werden.

3.5.2.1 Wird ein normales Integer-Bitfeld als Bitfeld vom Datentyp *signed int* oder *unsigned int* behandelt?

Normale Integer-Bitfelder werden als **signed int** behandelt.

3.5.2.1 Reihenfolge, in der innerhalb eines *int* Bitfelder angeordnet werden.

Von der niederwertigsten Bitposition aufsteigend zur höchstwertigen Bitposition.

3.5.2.1 Auffüllen und Ausrichten der Elemente von Strukturen.

Standardgemäß wird nicht aufgefüllt. Mit der Option *Ausrichtung an geraden Adressen* (*O/C/G/G*) für die wortweise Ausrichtung an Speicheradressen wird der Compiler veranlaßt, die Strukturen und jedes ihrer Elemente, das nicht den Datentyp **char** hat, jeweils ab einer geradzahligen Adresse zu speichern und dafür zu sorgen, daß der gesamte belegte Speicherbereich eine gerade Anzahl von Bytes umfaßt.

3.5.2.1 Darf ein Bitfeld die Begrenzung einer Speicherzelle überschreiten?

Wenn nicht explizit Ausrichtung an geradzahligen Speicheradressen verlangt wird (*O/C/G/G*), können Bitfelder Wortgrenzen überschreiten – sie werden aber niemals in mehr als 2 aufeinanderfolgenden Bytes gespeichert.

3.5.2.2 Welcher Integer-Typ wird gewählt, um die Werte eines Aufzählungstyps darzustellen?

Wenn alle Elemente eines Aufzählungstyps in ein **unsigned char** passen, dann wird dieser Datentyp gewählt, ansonsten der Datentyp **signed int**.

3.5.3 Was konstituiert den Zugriff auf ein Objekt, das einen als *volatile* deklarierten Datentyp hat?

Jeder Bezug auf ein solches als **volatile** deklariertes Objekt wird auf dieses zugreifen. Ob ein Zugriff auf benachbarte Speicherbereiche auch mit dem Zugriff

auf das Objekt verbunden ist, hängt vom Speicheraufbau des Rechners ab. Für spezielle Gerätespeicher wie den Bildschirmspeicher, hängt dies von der Konstruktion der Hardware ab. Beim normalen PC-Speicher werden **volatile**-Objekte nur für den Speicher verwendet, der von asynchronen Interrupts angesprochen werden könnte. Daher hat der Zugriff auf benachbarte Objekte keine Auswirkungen.

3.5.4 Maximalzahl von Deklaratoren, die einen arithmetischen Datentyp, Struktur- oder Varianten-Datentyp modifizieren können.

Es ist keine besondere Begrenzung festgelegt. Die Zahl der erlaubten Deklaratoren ist ziemlich groß, jedoch wird sie reduziert, wenn die Deklaratoren in tief verschachtelten Anweisungsblöcken einer Funktion stehen. Die erlaubte Anzahl auf Dateiebene ist wenigstens 50.

3.6.4.2 Maximale Anzahl von case-Werten in einer switch-Anweisung.

Es ist keine besondere Begrenzung festgelegt. Solange genügend Speicherplatz zur Verfügung steht, wird der Compiler **case**-Werte akzeptieren.

3.8.1 Stimmt der Wert der Zeichenkonstanten in einem Ausdruck, der Bedingungs-Anweisungen kontrolliert, mit dem Wert derselben Zeichenkonstanten im Zeichensatz des ausführbaren Codes überein? Kann eine solche Zeichenkonstante auch einen negativen Wert haben?

Alle Zeichenkonstanten, auch die in bedingten Compilerdirektiven, benutzen denselben Zeichensatz im Quelltext und im ausführbaren Code. Zeichenkonstanten können negative Werte haben, wenn ihr Datentyp **signed** ist und diese Voreinstellung des Compilers nicht durch die Option *O/C/G/V* verändert wurde.

3.8.2 Methode zum Lokalisieren von Include-Quelltextdateien.

Bei Header-Dateinamen in spitzen Klammern wird die jeweilige Datei in jedem der Include-Verzeichnisse gesucht.

Sind keine Suchwege festgelegt, wird nur das aktuelle Verzeichnis durchsucht.

3.8.2 Werden bei Include-Quelltextdateien Namen in Anführungszeichen unterstützt?

Ja. In diesem Fall durchsucht Turbo C++ erst das aktuelle Verzeichnis nach der angegebenen Include-Datei. War dies erfolglos, wird danach genauso wie bei Header-Dateien in spitzen Klammern vorgegangen.

3.8.2 Die Abbildung von Zeichenfolgen der Quelltext-Dateinamen.

Backslashes in den Namen von Include-Datei werden als einzelne Zeichen, nicht als Escape-Zeichen behandelt. Groß- und Kleinschreibung wird ignoriert.

3.8.8 Definition von __DATE__ und __TIME__, wenn sie nicht verfügbar sind.
Datum und Uhrzeit sind immer verfügbar und verwenden Systemdatum und -uhrzeit von DOS.

4.1.1 Zeichen für den Dezimalpunkt.
Ist ein Punkt (.).

4.1.5 Datentyp des *sizeof*-Operators, *size_t*.
Der Datentyp von *size_t* ist **unsigned int**.

4.1.5 Null-Zeigerkonstante, zu der das Makro NULL erweitert wird.
int 0 oder **long** 0, je nach Speichermodell.

4.2 Ausgegebene Diagnose und das Abbruchverhalten der Funktion *assert*.
Es wird die Meldung »`Assertion failed: Ausdruck file Dateiname line Zeilennummer` « ausgegeben. *Ausdruck* ist die Bedingung, die nicht eingetreten ist und daher den der Meldung unmittelbar folgenden Programmabbruch verursacht. *Zeilennummer* und *Dateiname* stehen für die Zeile und Quelltextdatei, in der die Funktion *assert* aufgerufen wurde.
Die Funktion *abort* wird sofort nach Ausgabe der Meldung aufgerufen und bricht das Programm ab.

4.3 Bei der Implementation definierte Aspekte für das Testen von Zeichen und von Funktionen, die Klein- und Groß-schreibung aufeinander abbilden.
Keine vorhanden außer den unter 4.3.1 erwähnten.

4.3.1 Der Zeichensatz, der durch die Funktionen *isalnum*, *isalpha*, *iscntrl*, *islower*, *isprint* und *isupper* geprüft wird.
Die ersten 128 ASCII-Zeichen.

4.5.1 Werte, die von mathematischen Funktionen bei einem Definitionsbereich-Fehler zurückgeliefert werden.
Ein IEEE NAN (»not a number« =keine Zahl).

4.5.1 Setzen mathematische Funktionen bei Unterlauf-Fehlern den Integer-Ausdruck *errno* auf den Wert des Makros ERANGE?
Nein, nur bei den anderen Fehlern – Definitionsbereich, Singularität, Überlauf und Genauigkeitsfehlern.

4.5.6.4 Ergibt sich ein Definitionsbereich-Fehler oder wird eine Null zurückgeliefert, wenn die Funktion *fmod* als zweites Argument 0 erhält?
Nein. *fmod*(*x*,*0*) liefert 0.

4.7.1.1 Signale für die Funktion *signal*.
SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM.

4.7.1.1 Semantik der Signale, die von *signal* erkannt werden.
Vergleiche die Beschreibung von **signal** im Hilfesystem.

4.7.1.1. Standardhandhabung und die Behandlungsroutine beim Programmstart für die Signale, die von *signal* erkannt werden.
Vergleiche die Beschreibung von **signal** im Hilfesystem.

4.7.1.1. Wenn das Äquivalent zu *signal*(*sig, SIG_DFL*); nicht vor dem Aufruf eines signal-Handlers ausgeführt wird, wie wird dann das Blockieren des Signals durchgeführt?.
Das Äquivalent zu *signal*(*sig, SIG_DFL*); wird immer ausgeführt.

4.7.1.1. Wird die Standardroutine auf den Vorgabewert zurückgestellt, wenn das SIGILL-Signal von einer Routine empfangen wird, die für die *signal*-Funktion definiert worden ist?
Nein.

4.9.2 Muß die letzte Zeile eines Text-Streams mit einem Zeichen für Zeilenvorschub beendet werden?
Nein, es wird kein Zeichen benötigt.

4.9.2 Stehen Leerzeichen, die unmittelbar vor einem Zeichen für Zeilenvorschub in einen Text-Stream geschrieben wurden, auch beim Einlesen wieder vor diesem Zeichen?
Ja.

4.9.2 Wieviele Null-Zeichen dürfen Daten folgen, die in einen binären Stream geschrieben werden?
Überhaupt keine.

4.9.3 Wird der Dateipositions-Zeiger für einen Stream im APPEND-Modus anfänglich auf den Dateibeginn oder das Dateiende gesetzt?
Der Positionsindikator wird anfänglich auf den Dateibeginn gesetzt. Jeder Schreibvorgang setzt den Dateipositions-Zeiger automatisch auf das Dateiende.

4.9.3 Schneidet das Schreiben in einen Text-Stream die mit diesem verbundene Datei ab diesem Punkt ab?.
Das Schreiben von 0 Bytes schneidet die Datei *nicht notwendigerweise* ab. Es kommt darauf an, wie die Datei gepuffert ist. Es ist sicherer, die Folgen einer Schreibaktion mit der Länge Null als unvorhersehbar zu klassifizieren.

4.9.3 Die Eigenschaften der Dateipufferung.
Dateien können ganz, zeilenweise oder überhaupt nicht gepuffert werden. Wenn eine Datei gepuffert wird, wird standardmäßig ein Puffer von 512 Bytes beim Öffnen der Datei angelegt.

4.9.3 Existiert eine Datei mit der Länge 0?
Ja.

4.9.3 Kann dieselbe Datei mehrfach geöffnet werden?
Ja.

4.9.4.1 Welche Auswirkung hat die Funktion *remove* auf eine offene Datei?
Es gibt keine speziellen Überprüfungen für eine bereits geöffnete Datei. Der Programmierer muß sich darum kümmern.

4.9.4.2 Welche Wirkung hat der Aufruf der Funktion *rename*, wenn der neue Dateiname bereits existiert?
rename liefert den Wert -1, und *errno* wird auf EEXIST gesetzt.

4.9.6.1 Ausgabe bei der Verwendung von %*p*-Umwandlung in *fprintf*.
In **near**-Datenmodellen vier Hexstellen (XXXX). In **far**-Datenmodellen vier Hexstellen, Doppelpunkt und weitere vier Hexstellen (XXXX:XXXX).

4.9.6.2 Eingabe bei der Verwendung von %*p*-Umwandlung in *fscanf*.
Vgl. hierzu 4.9.6.1.

4.9.6.2 Behandlung des Zeichens – (Bindestrich), wenn es weder das erste noch das letzte Zeichen in der Scan-Liste für eine %[-Umwandlung in *fscanf* ist.
Vgl. hierzu die Erläuterung von *scanf* im Hilfesystem.

4.9.9.1 Auf welchen Wert wird der Ausdruck *errno* gesetzt, wenn *fgetpos* oder *ftell* einen Fehler liefern?
EBADF (Bad File Number).

4.9.10.4 Von *perror* erzeugte Fehlermeldungen.

Meldung	Bedeutung
Error 0	»kein Fehler«
Invalid function number	ungültige Subfunktionsnummer
No such file or directory	Datei/Suchweg nicht gefunden
Path not found	Suchweg nicht gefunden
Too many open files	Maximalzahl von Dateien bereits offen
Permission denied	Zugriff nicht gestattet

Meldung	Bedeutung
Bad file number	Handle nicht definiert/Datei nicht offen
Memory area trashed	Speicher-Kontrollblocks zerstört
Not enough memory	nicht genug Platz im Hauptspeicher
Invalid memory block address	ungültige Speicherblock-Startadresse
Invalid environment	ungültiges *Environment*
Invalid format	ungültiges Format
Invalid access code	ungültiger Wert für *access*

Vgl. auch *perror* im Hilfesystem für weitere Details.

4.10.3 Verhalten von *calloc*, *malloc* oder *realloc*, wenn die gewünschte Größe 0 ist.

calloc und *malloc* werden diese Anforderung ignorieren, *realloc* wird den Speicherblock freigeben.

4.10.4.1 Verhalten von *abort* in Bezug auf offene und temporäre Dateien.

Der Inhalt der Dateipuffer wird nicht geschrieben und die Dateien werden nicht geschlossen.

4.10.4.3 Status, der von *exit* zurückgeliefert wird, wenn der Wert des Arguments nicht Null, EXIT_SUCCESS oder EXIT_FAILURE ist.

Kein besonderer. Es wird der Status geliefert, der übergeben wurde. Der Status wird als **signed char** dargestellt.

4.10.4.4 Menge von Environment-Namen und die Methode zum Anpassen der Environment-Tabelle, die von *getenv* verwendet wird.

Die *Environment*-Namen sind die Einträge, die mit dem SET-Befehl in DOS definiert wurden. Diese werden in der von DOS unterhaltenen Tabelle *Environment* gespeichert. *putenv* kann zum Ändern dieser Einträge für die Dauer eines laufenden Programms verwendet werden. Der DOS-Befehl SET muß benutzt werden, um einen *Environment*-String permanent zu verändern.

4.10.4.5 Inhalt und Art der Ausführung des Strings, der durch die Funktion *system* ausgeführt wird.

Der String wird als DOS-Befehl behandelt. COMMAND.COM wird geladen, und der String wird als Parameter übergeben und ausgeführt. Jeder DOS-Befehl, jede Batch-Datei und jedes Programm können ausgeführt werden.

4.11.6.2 Inhalt der Fehlermeldungen, die von *strerror* zurückgegeben werden.

Vgl. 4.9.10.4.

4.12.1 Orts- (Zeitzonen) und Sommerzeit.
Definiert als lokale PC-Zeit und Datum.

4.12.2.1. Einheit von *clock*.
clock liefert die seit dem Start des Programms vergangene Zeit in »Clock-Ticks« zurück.

4.12.3.5 Formate von Datum und Uhrzeit.
Turbo C++ implementiert ANSI-Formate.

10. Fehlermeldungen

Dieser Anhang befaßt sich mit folgenden Meldungen von Turbo C++:

- Compiler-Meldungen;
- Hilfe-Meldungen;
- Laufzeit-Meldungen;
- Linker-Meldungen;
- Bibliotheksverwaltungs-Meldungen;

Der *Typ* der jeweiligen Meldung (beispielsweise Compiler oder Laufzeit) sowie die Art des Fehlers (schwerer Fehler, Fehler oder Warnung) wird in Klammern vor der eigentlichen Erläuterung angegeben. In den meisten Erläuterungen werden mögliche Ursache und Abhilfe für die Fehlermeldung bzw. Warnung angeführt.

Auffinden einer Fehlermeldung

Die Meldungen sind in alphabetischer ASCII-Reihenfolge angeordnet. Meldungen, die mit Symbolen beginnen (wie Gleichheitszeichen, Komma, Klammern, etc.), sind als erste aufgeführt, gefolgt von Zahlen und Buchstaben.

Da Meldungen, die mit einer Variablen beginnen, nicht alphabetisch eingeordnet werden können, werden diese nach dem der Variablen folgenden Wort eingeordnet. Wenn Sie z.B. eine C++ Funktion *goforit* definiert haben und nach dem Compilieren die Fehlermeldung

```
goforit must be declared with no arguments
```

erhalten, suchen Sie in der Liste der Fehlermeldungen unter dem Buchstaben »m« wie »must« und finden dann folgenden Eintrag:

```
Funktion must be declared with no arguments
```

Wenn eine Variable nicht am Anfang einer Fehlermeldung steht (beispielsweise »Address of overloaded function *Funktion* doesn't match *Type*«), finden Sie diese Meldung an der entsprechenden alphabetischen Position, in dem angeführten Fall also unter dem Buchstaben *A*.

Arten von Fehlermeldungen

Die Fehlermeldungen, die Sie erhalten, sind unterschiedlicher Natur und hängen davon ab, woher sie stammen. Im folgenden Abschnitt wird jede Fehlerart zusammen mit einer Aufstellung der möglichen Variablen erläutert.

Compiler-Meldungen

Die Compiler-Fehlermeldungen von Turbo C++ werden in drei Klassen unterteilt: Schwere Fehler, Fehler und Warnungen.

Fatal Error (schwerer Fehler). Diese Fehler treten nicht sehr häufig auf. Einige dieser Meldungen deuten auf einen internen Compilerfehler hin. Schwere Fehler führen zu einem sofortigen Abbruch der Compilierung. Sie müssen dann den Fehler korrigieren und die Compilierung erneut starten.

Error (Fehler). Meldungen dieser Art deuten auf syntaktisch inkorrekte Befehle und auf Fehler in Kommandozeileneingaben sowie bei Disketten- oder Plattenbzw. Speicherzugriffen hin. Der Compiler schließt die aktuellen Übersetzungsphase ab und bricht dann ab. Der Compiler versucht, möglichst viele wirkliche Fehler während jeder Compilierungsphase (Präprozessor, »Parsing«, Optimierung und Code-Erzeugung) im Quelltext zu finden.

Warning (Warnung). Warnungen führen nicht zum Abbruch der Compilierung. Sie zeigen aber an, daß dem Compiler eine Befehlsfolge verdächtig vorkommt, die jedoch nicht gegen die Sprachregeln verstößt. Der Compiler gibt Warnungen auch dann aus, wenn Sie maschinenabhängige Konstrukte in Ihren Quelltextdateien verwenden.

Bei der Ausgabe der Fehler wird zuerst die Fehlerkategorie, dann der Name der Quelltextdatei und die Nummer der Zeile, in der der Compiler den Fehler entdeckt hat, aufgelistet. Danach folgt der Text der Fehlermeldung.

Die ausgegebenen Zeilennummern sind nicht exakt.
Sie sollten sich bei den angeführten Zeilennummern in Fehlermeldungen folgender Besonderheit bewußt sein: der Compiler erzeugt Meldungen, sobald er auf eine Fehlerbedingung stößt. Da die Sprachen C und C++ keine Restriktionen kennen, wie Anweisungen auf eine Textzeile zu verteilen sind, kann die eigentliche Fehlerursache einige Zeilen vor oder nach der in der Meldung ausgewiesenen Zeilennummer stehen.

Folgende generische Namen (größtenteils selbsterklärend) und Werte werden in der Auflistung der Meldungen in diesem Kapitel verwendet; beim Auftreten einer entsprechenden Fehlerbedingung werden sie durch die jeweils entsprechenden Werte oder Namen ersetzt.

Tabelle 10.1 Variablen in Compiler-Meldungen

Im Handbuch	Auf dem Bildschirm
Argument	Ein Kommandozeilen- oder Funktionsargument
Befehl	Ein DOS-Befehl
Bezeichner	Der Name einer Variable etc.
Dateiname	Ein Dateiname (mit oder ohne Namenserweiterung)
Element	Der Name eines Datenelements oder einer Elementfunktion
Funktion	Ein Funktionsname
Gruppe	Ein Gruppenname
Klasse	Ein Klassenname
Makro	Ein Makroname
Meldung	Ein Meldungsstring
Nummer	Eine Zahl
Option	Eine Kommandozeilenoption oder eine andere Option
Parameter	Ein Parametername
Spezifizierer	Ein Typ-Spezifizierer
Sprache	Der Name einer Programmiersprache
Struktur	Der Name eines struct-Objektx
Symbol	Ein Symbolname
Typ	Ein Typname
Zeichen	Ein Zeichen
Zeilennummer	Eine Zeichennummer einer Quelltextdatei

Hilfe-Compiler-Meldungen

Der Hilfe-Compiler gibt Meldungen aus, wenn er auf Fehler im Aufbau der Ressourcendatei trifft. Fehler während der Bearbeitung der Projektdatei werden mit dem Buchstaben *P* eingeleitet und z.B. folgendermaßen ausgegeben:

```
Error P1025: line...7 of filename.HPJ : Section heading
sectionname unrecognized.

Warning P1039: line...38 of filename.HPJ : [BUILDTAGS]
section missing.
```

Fehler, die bei der Bearbeitung der RTF-Themendatei(en) auftreten, werden mit dem Buchstaben *R* eingeleitet und z.B. folgendermaßen ausgegeben:

```
Error R2025: File environment error.
Warning R2501: Using old key-phrase table.
```

Themennummern

Der Compiler zeigt – wenn es möglich ist – die Nummer des Themas und den Namen der Datei an, die den Fehler enthält. Da Themen nicht numeriert sind, bezieht sich die bei einer Fehlermeldung angezeigte Nummer auf die sequentielle Position des Themas in der RTF-Datei (erstes, zweites, etc.). Diese Nummern können identisch mit der Seitennummer sein, die Ihr Textprogramm

anzeigt (abhängig von der Zeilenanzahl, die Sie einer angenommenen Druckseite zugewiesen haben). Denken Sie daran, daß die einzelnen Themen mit harten Seitenumbrüchen voneinander getrennt werden, auch wenn es im Hilfesystem keine Seiten gibt.

Meldungen, die mit dem Wort »Error« beginnen, sind schwere Fehler. Schwere Fehler werden immer gemeldet. Bei deren Auftreten ist die resultierende Ressourcendatei fehlerhaft. Meldungen, die mit dem Wort »Warning« beginnen, sind weniger gravierend. Werden während der Compilierung Warnungen ausgegeben, entsteht trotzdem eine gültige Ressourcendatei, die unter Windows geladen werden kann. Es können jedoch unter Umständen Fehler darin vorhanden sein. Sie können die Anzahl der Warnungen festlegen, die vom Compiler ausgegeben werden sollen.

Während die Projektdatei abgearbeitet wird, ignoriert der Compiler Zeilen, die Fehler enthalten, und versucht, den Prozeß aufrechtzuerhalten. Das hat zur Folge, daß Fehler, die zeitig in einer Compilierung auftreten, im Verlauf des Prozesses sehr viele weitere Fehlermeldungen bedingen. Ähnlich werden die während des Aufbaus der RTF-Themendateien auftretenden Fehler gemeldet. Wenn es sich dabei nicht um schwere Fehler handelt, setzt der Compiler den Aufbauprozeß fort.

Eine einzige Fehlerbedingung in der Themendatei kann mehrere Fehlermeldungen des Compilers nach sich ziehen. Beispielsweise verursacht ein falsch gekennzeichnetes Thema einen Fehler, der immer dann gemeldet wird, wenn sich ein Sprungthema auf den korrekten Themenbezeichner bezieht. Dieser Fehler kann leicht behoben werden, indem die Fußnote, die den falschen Kontextstring enthält, korrigiert wird.

Tabelle 10.2 Variablen in Hilfe-Compiler-Meldungen

Im Handbuch	Auf dem Bildschirm
Abschnittname	Ein Abschnittname
Dateiname	Ein Dateiname mit oder ohne Namenserweiterung
Kontextname	Ein Kontextname
Optionsname	Ein Optionsname
Schriftname	Ein Schriftname
Tag-Name	Ein symbolischer Name (Build Tag)
Themennummer	Eine Themennummer

Laufzeit-Meldungen

In Turbo C++ gibt es nur wenige Laufzeit-Fehlermeldungen. Diese werden von einem erfolgreich compilierten und gelinkten Programm zur Laufzeit ausgegeben, wenn eine entsprechende Fehlerbedingung eintritt.

Meldungen der Bibliotheksverwaltung

Die Bibliotheksverwaltung verfügt über »Fehlermeldungen« und »Warnungen«. Die in der folgenden Tabelle aufgeführten generischen Namen und Werte erscheinen in den Bibliotheksverwaltungs-Meldungen. Sie werden in der aktuellen Meldung durch die entsprechenden Namen und Werte ersetzt:

Tabelle 10.3 Variablen in Meldungen der Bibliotheksverwaltung

Im Handbuch	Auf dem Bildschirm
Dateiname	Ein Dateiname mit oder ohne Namenserweiterung.
Größe	Eine Zahl
Länge	Eine Zahl
Modul	Ein Modulname
Nummer	Eine Zahl
Pfad	Ein Pfad
Symbol	Ein Symbolname
Typ	Ein Typname
Ursache	Eine in der Meldung angegebene Fehlerursache

Linker-Meldungen

Der Linker kennt drei Arten von Meldungen: »Schwere Fehler«, »Fehler« und »Warnungen«.

- Ein »schwerer Fehler« hat den sofortigen Abbruch von des Linkers zur Folge; die .EXE-Datei wird gelöscht.
- Ein »Fehler« löscht zwar die .EXE- und .MAP-Dateien nicht, Sie sollten aber trotzdem die .EXE-Datei nicht ausführen. In der IDE werden Fehler wie schwere Fehler behandelt.
- »Warnungen« warnen Sie vor bestimmten Umständen, die Sie eventuell korrigieren sollten. .EXE- und .MAP-Dateien werden trotz einer Warnung erzeugt.

Die folgenden generischen Namen und Werte erscheinen in Linker-Meldungen. Sie werden in der aktuellen Meldung durch die entsprechenden Namen und Werte ersetzt:

Tabelle 10.4 Variablen in den Meldungen des Linkers

Im Handbuch	Auf dem Bildschirm
Dateiname	Ein Dateiname mit oder ohne Namenserweiterung
Fehlernummer	Fehlercode bei internen Fehlern
Gruppe	Ein Gruppenname

Im Handbuch	Auf dem Bildschirm
Modul	Ein Modulname
Nummer	Eine Zahl
Segment	Ein Segmentname
Symbol	Ein Symbolname
XXXXh	Eine vierstellige hexadezimale Zahl, gefolgt von einem *h*
Zeilennummer	Eine Zeilennummer in einer Datei

Erläuterung der Meldungen

(expected
[(erwartet]
(Compiler, Fehler) Vor einer Parameterliste wird (erwartet.

) expected
[) erwartet]
(Compiler, Fehler) Am Ende einer Parameterliste wird) erwartet.

, expected
[, erwartet]
(Compiler, Fehler) In einer Liste von Deklarationen, Initialisierungen oder Parametern wird ein Komma erwartet.

: expected after private/protected/public
[: *nach private/protected/public* erwartet]
(Compiler, Fehler) Am Beginn eines entsprechenden Abschnitts einer C++ Klasse muß auf das jeweilige Schlüsselwort (**private**, **protected** oder **public**) ein Doppelpunkt (:) folgen.

< expected
[< erwartet]
(Compiler, Fehler) Auf das Schlüsselwort **template** folgt nicht das Zeichen <. Jede Template-Deklaration muß die formalen Template-Parameter unmittelbar nach dem Schlüsselwort **template** in spitzen Klammern <> enthalten.

{ expected
[{ erwartet]
(Compiler, Fehler) Beim Beginn eines Blocks oder einer Initialisierung wird { erwartet.

} expected
[} erwartet]
(Compiler, Fehler) Am Ende eines Blocks oder einer Initialisierung wird } erwartet.

Abnormal program termination
[Außergewöhnliches Programmende]
(Laufzeit, Fehler) Diese Meldung kann erzeugt werden, wenn dem Programm nicht genügend Speicher zur Verfügung steht. Die Meldung wird von *abort* erzeugt und durch Speicherüberschreibungen ausgelöst.

Access can only be changed to public or protected
[Zugriff nur in **public** oder **protected** veränderbar]
(Compiler, Fehler) Eine abgeleitete Klasse in C++ kann die Zugriffsrechte eines Basiselements der Basisklasse nur in **public** oder **protected** verändern. Ein Element einer Basisklasse kann nicht in **private** verändert werden.

Address of overloaded function *Funktion* doesn't match *Typ*
[Adresse der überladenen Funktion *Funktion* paßt nicht zum *Typ*]
(Compiler, Fehler) Die Adresse einer überladenen Funktion wurde einer Variablen zugewiesen bzw. ein Parameter wurde damit initialisiert. Der Typ der Variablen bzw. des Parameters paßt jedoch zu keiner der überladenen Funktionen mit dem angegebenen Namen.

Ambiguity between *Funktion1* and *Funktion2*
[*Funktion1* und *Funktion2* mehrdeutig]
(Compiler, Fehler) Beide überladenen Funktionen könnten über die angegebenen Parameter angesprochen werden. Diese Mehrdeutigkeit ist nicht zulässig.

Ambiguous member name *Name*
[Elementname *Name* mehrdeutig]
(Compiler, Fehler) Der Name von Elementen in Strukturen in Inline-Assembler-Anweisungen muß eindeutig sein. Ist er in mehreren Strukturen definiert, müssen alle Definitionen in Bezug auf Typ und Offset übereinstimmen. Der Elementname ist in diesem Fall mehrdeutig. Verwenden Sie die Syntax *(struct xxx).yyy*.

Ambiguous operators need parentheses
[Mehrdeutige Operatorkombination ohne Klammerung]
(Compiler, Warnung) Diese Warnung erhalten Sie bei der Kombination von relationalen, bitweisen oder Schiebe-Operatoren ohne entsprechende Klammerung oder bei der Kombination von Schiebeoperationen mit Addition und Subtraktion. Meist liegt dieser Warnung eine Verwechslung der Rangfolge der Operatoren durch den Programmierer zugrunde.

Array allocated using new may not have an initializer
[Das mit *new* reservierte Array darf keinen Initialisierer verwenden]
(Compiler, Fehler) Bei der Initialisierung eines Vektors (Array) von Klassen muß der Konstruktor verwendet werden, der ohne Argumente arbeitet. Dieser Konstruktor wird *Standardkonstruktor* genannt. Bei der Initialisierung eines derartigen Vektors ist die Angabe von Konstruktorargumenten nicht zulässig.

Array bounds missing]
[] bei Array-Deklaration oder Adressierung erwartet]
(Compiler, Fehler) In Ihrem Quelltext ist ein Array deklariert, dessen rechte eckige Klammer fehlt.

Array must have at least one element
[Array muß mindestens ein Element enthalten]
(Compiler, Fehler) ANSI C und C++ schreiben vor, daß ein Array mindestens ein Element enthalten muß, da Objekte der Größe Null nicht erlaubt sind. Ein alter Programmiertrick sieht als Element einer Struktur ein Array der Größe Null vor und liefert später den nötigen Speicherplatz mit *malloc*. Um diesen Trick hier anzuwenden, müssen Sie mindestens ein Element für das Array vorsehen. Im Gegensatz zu Definitionen sind nach wie vor Deklarationen von Arrays unbekannter Größe möglich, wie z.B.:

```
char ray[];         /*Definition unbekannter Größe - nicht erlaubt*/
char ray[0];        /*Definition der Größe Null - nicht erlaubt*/
extern char ray[];  /*Deklaration unbekannter Größe - erlaubt*/
```

Array of references is not allowed
[Array aus Referenzen nicht erlaubt]
(Compiler, Fehler) Ein Array aus Referenzen ist nicht zulässig, weil Zeiger auf Referenzen nicht erlaubt sind und Array-Namen in Zeiger umgewandelt werden.

Array size for 'delete' ignored
[Array-Größe für *delete* wird ignoriert]
(Compiler, Warnung) In der neuen Version von C++ ist die Angabe der Array-Größe beim Löschen des Array nicht mehr nötig. Der Compiler ignoriert die Angabe und gibt diese Warnung aus, wenn Sie ältere Quelltexte compilieren.

Array size too large
[Array zu groß]
(Compiler, Fehler) Das deklarierte Array ist größer als 64 KByte.

Array variable *Bezeichner* is near
[Array-Variable *Bezeichner* ist vom Typ **near**]
(Compiler, Warnung) Bei der Auswahl von *Optionen/Compiler/Erweiterte Code-Generierung.../Automatische Far Daten* definiert der Compiler globale Variablen, die größer als der Schwellenwert sind, automatisch als **far**. Ist die Variable aber ein initialisiertes Array mit nicht festgelegter Größe, ist dem Compiler die Gesamtgröße nicht bekannt, wenn er eine Entscheidung für **near** oder **far** treffen muß. Der Compiler entscheidet sich in diesem Fall für **near**. Wenn die Gesamtgröße der Variablen auf Grund der Anzahl der Initialisierer den Schwellenwert überschreitet, gibt der Compiler diese Meldung aus. Wenn durch die Tatsache, daß der Compiler die Variable als **near** festlegt, Probleme entstehen (z.B. der Linker Gruppenüberlauf auf Grund zu umfangreicher gobaler Daten meldet), müssen Sie die betreffende Variable explizit als **far** deklarieren. Dazu fügen Sie das Schlüsselwort **far** unmittelbar vor dem Variablennamen in die Definition ein.

Assembler statement too long
[**asm**-Anweisung zu lang]
(Compiler, Fehler) Inline-Assembler-Anweisungen dürfen nicht mehr als 480 Bytes umfassen.

Assigning *Typ* **to** *enumeration*
[Zuweisung von *Typ* an **enum**]
(Compiler, Warnung) Es ist ein Fehler in C++, einem **enum**-Typ einen Integerwert zuzuweisen, hier wird aber nur eine Warnung ausgegeben, damit existierende Programme laufen können.

Assignment to this not allowed, use X::operator new instead
[Zuweisung an den Operator **this** ist nicht zulässig, verwenden Sie *X::operator new*]
(Compiler, Fehler) In früheren C++ Versionen konnte man die Speicherbelegung von Klassenobjekten nur über Zuweisung an den Parameter **this** eines Konstruktors steuern. Dies ist jetzt nicht mehr erlaubt, da es besser und sicherer ist, eine *operator new*-Elementfunktion zu definieren.

Attempt to export non-public symbol *Symbol*
[Das nicht-öffentliche Symbol *Symbol* sollte exportiert werden]
(Linker, Warnung) Im EXPORTS-Bereich einer Moduldefinitionsdatei war ein Symbolname aufgeführt, aber in den gelinkten Modulen wurde kein öffentliches Symbol mit diesem Namen gefunden. Sie haben entweder einen Tippfehler gemacht oder die Schreibweise (groß/klein) nicht beachtet, bzw. es wurde überhaupt keine Funktion mit diesem Namen definiert.

Attempt to grant or reduce access to *Bezeichner*
[Versuch, den Zugriff auf einen Bezeichner zu gewähren oder zu verhindern]
(Compiler, Fehler) Eine abgeleitete Klasse von C++ kann die Zugriffsrechte eines Basisklassenelements ändern. Es ist aber nur die Wiederherstellung seiner Rechte in der Basisklasse erlaubt, das Hinzufügen oder Entfernen von Zugriffsrechten ist nicht gestattet.

Attempting to return a reference to a local object
[Rückgabe einer Referenz auf ein lokales Objekt]
(Compiler, Fehler) Sie haben in einer Funktion, die einen Referenztyp zurückliefert, versucht, eine Referenz auf ein temporäres Objekt zurückzuliefern (eventuell das Ergebnis eines Konstruktor- oder Funktionsaufrufs). Da das Objekt beim Rücksprung der Funktion freigegeben wird, wird auch die Referenz ungültig.

Attempting to return a reference to local variable *Bezeichner*
[Rückgabe einer Referenz auf lokale Variable *Bezeichner* versucht]
(Compiler, Fehler) Diese C++ Funktion liefert einen Referenztyp und Sie versuchen, eine Referenz auf eine lokale (**auto**) Variable zurückzuliefern. Dies ist nicht zulässig, weil die lokale Variable beim Verlassen der Funktion freigegeben wird. Eine Referenz darf sich nur auf eine statische oder globale Variable beziehen, oder die Funktion muß einen Wert anstatt einer Referenz liefern.

Automatic data segment exceeds 64K
[Das automatische Datensegment umfaßt mehr als 64 KByte]
(Linker, Fehler) Der Gesamtumfang des physikalischen Segments DGROUP, des lokalen Heap und Stack überschreitet die 64-KByte-Grenze. Geben Sie entweder in den Anweisungen HEAPSIZE und STACKSIZE in der Moduldefinitionsdatei niedrigere Werte ein, oder verringern Sie die Größe der **near**-Daten in DGROUP. In der .MAP-Datei werden die Größen der Segmente in DGROUP angezeigt. Die Map-Datei zeigt an, wieviel jedes Modul zu DGROUP beiträgt.

Bad call of intrinsic function
[Unzulässiger Aufruf einer **intrinsic**-Funktion]
(Compiler, schwerer Fehler) Sie haben eine **intrinsic**-Funktion verwendet, ohne einen Prototyp bereitzustellen, oder Sie haben einen Prototyp für eine **intrinsic**-Funktion angegeben, den der Compiler nicht erwartet hat.

Bad character in parameters
[Unzulässiges Zeichen in der Kommandozeile oder in einer Response-Datei]
(Linker, schwerer Fehler) Eines der folgenden Zeichen wurde in der Kommandozeile oder in einer Response-Datei gefunden:

" * <=>? [ ] |

Es kann aber auch sein, daß ein unzulässiges Steuerzeichen aufgetreten ist. Als Steuerzeichen sind erlaubt: horizontaler Tabulator, Line Feed, Carriage Return und Strg Z .

Bad define directive syntax
[Ungültige Syntax einer *define*-Anweisung]
(Compiler, Fehler) Eine Makrodefinition beginnt oder endet mit dem ##-Operator oder enthält den #-Operator, auf den aber kein Parametername folgt.

Bad file name format in include directive
[Ungültiges Dateinamensformat in einer #*include*-Anweisung]
(Compiler, Fehler) Dateinamen für #*include* müssen entweder von Anführungszeichen ("filename.h") oder spitzen Klammern (<filename.h>) eingefaßt sein. Diese Fehlermeldung erhalten Sie, wenn das erste Anführungszeichen oder die linke spitze Klammer fehlt. Die Meldung erscheint auch dann, wenn ein Makro verwendet wurde und der daraus resultierende Text nicht korrekt ist, d.h. wenn er nicht in Anführungszeichen steht.

Bad file name format in line directive
[Ungültiges Dateinamensformat in line-Anweisung]
(Compiler, Fehler) Dateinamen für #*include* müssen entweder von Anführungszeichen ("filename.h") oder spitzen Klammern (<filename.h>) eingefaßt sein. Diese Fehlermeldung erhalten Sie, wenn das erste Anführungszeichen oder die linke spitze Klammer fehlt. Die Meldung erscheint auch dann, wenn ein Makro verwendet wurde und der daraus resultierende Text nicht korrekt ist, d.h. wenn er nicht in Anführungszeichen steht.

Bad GCD type in GRPDEF, extended dictionary aborted
[Ungültiger GCD-Typ in GRPDEF, erweitertes Verzeichnis wird nicht angelegt]
(Bibliotheksverwaltung, Warnung) Die Bibliotheksverwaltung hat bei der Erstellung eines erweiterten Verzeichnisses einen ungültigen Eintrag in der Gruppendefinition (GRPDEF) eines Objektmoduls gefunden.

Die Bibliotheksverwaltung (und der Linker) unterstützen nur Segement-Index-Typen. Ist ein anderer Typ für GRPDEF angegeben, kann die Bibliotheksverwaltung kein erweitertes Verzeichnis anlegen. Es ist möglich, daß ein Objektmodul, das nicht mit einem Borland-Programm entwickelt wurde, GRPDEF-Records anderer Typen erstellen kann. Diese Warnung wird auch dann ausgegeben, wenn ein beschädigtes Objektmodul auftritt.

Bad GRPDEF type encountered, extended dictionary aborted
[Ungültiger Typ in GRPDEF, erweitertes Verzeichnis wird nicht angelegt]
(Bibliotheksverwaltung, Warnung) Die Bibliotheksverwaltung hat bei der Erstellung eines erweiterten Verzeichnisses einen ungültigen Eintrag in der Gruppendefinition (GRPDEF) eines Objektmoduls gefunden.

Die Bibliotheksverwaltung (und der Linker) unterstützen nur Segement-Index-Typen. Ist ein anderer Typ für GRPDEF angegeben, kann die Bibliotheksverwaltung kein erweitertes Verzeichnis anlegen. Es ist möglich, daß ein Objektmodul, das nicht mit einem Borland-Programm entwickelt wurde, GRPDEF-Records anderer Typen erstellen kann. Diese Warnung wird auch dann ausgegeben, wenn ein beschädigtes Objektmodul auftritt.

Bad ifdef directive syntax
[Ungültige Syntax einer *ifdef*-Anweisung]
(Compiler, Fehler) Eine #*ifdef*-Anweisung darf ausschließlich einen einzelnen Bezeichner als Rumpf der Anweisung enthalten.

Bad object file record in library file ***Dateiname*** **in module** ***Modul*** **near module file offset 0** ***xxxxxxxx***
[Ungültiger Objektdateiabschnitt in Bibliotheksdatei *Dateiname* im Modul *Modul*, in der Nähe des Moduldateioffset 0*xxxxxxxx*]
(Linker, schwerer Fehler) Es liegt eine Objektdatei mit ungültigem Format vor. Die Ursache liegt in den meisten Fällen darin, daß eine Quelltextdatei oder eine Objektdatei benannt wird, die nicht vollständig aufgebaut wurde. Dies kommt vor, wenn der Computer während einer Compilierung gebootet wird oder wenn der Compiler seine Ausgabe-Objektdatei beim Drücken von Strg Untbr nicht löscht.

Bad object file record in module ***Dateiname*** **near module file offset 0** ***xxxxxxxx***
[Ungültiger Objektdateiabschnitt im Modul *Dateiname*, in der Nähe des Moduldateioffset 0*xxxxxxxx*]
(Linker, schwerer Fehler) Es liegt eine Objektdatei mit ungültigem Format vor. Die Ursache liegt in den meisten Fällen darin, daß eine Quelltextdatei oder eine Objektdatei benannt wird, die nicht vollständig aufgebaut wurde. Dies kommt

vor, wenn der Computer während einer Compilierung gebootet wird oder der Compiler seine Ausgabe-Objektdatei beim Drücken von Strg Untbr nicht löscht.

Bad OMF record type ***Typ*** **encountered in module** ***Modul***
[Ungültiger OMF-Recordtyp *Typ* in Modul *Modul*]
(Bibliotheksverwaltung, Fehler) Beim Lesen des Objektmoduls ist die Bibliotheksverwaltung auf einen ungültigen Objektmodulformat-Record (OMF) gestoßen. Die Bibiliotheksverwaltung hat den Kopf des Moduls *Modul* bereits gelesen und überprüft. Das bedeutet, daß das Objektmodul auf irgendeine Weise beschädigt wurde und neu erstellt werden muß.

Bad syntax for pure function definition
[Falsche Syntax in der Definition einer abstrakten Funktion]
(Compiler, Fehler) Abstrakte Funktionen erkennt man durch *=0* am Ende der Deklaration. Sie haben zwar etwas Ähnliches angegeben, aber nicht genau das Richtige.

Bad undef directive syntax
[Ungültige Syntax in #*undef*-Anweisung]
(Compiler, Fehler) Eine #*undef*-Anweisung darf ausschließlich einen einzelnen Bezeichner als Rumpf der Anweisung enthalten.

Bad version number in parameter block
[Falsche Versionsnummer im Parameterblock]
(Linker, schwerer Fehler) Der Fehler meldet eine interne Inkonsistenz in der IDE. Starten Sie die IDE erneut.

Base class ***Klasse*** **contains dynamically dispatchable functions**
[Basisklasse *Klasse* enthält dynamisch zugeordnete Funktionen]
(Compiler, Fehler) Im Augenblick unterstützen dynamische Zuweisungstabellen die Mehrfachvererbung nicht. Dieser Fehler tritt auf, weil eine Klasse, die eine DDVT-Funktion enthält, versucht hat, DDVT-Funktionen von mehreren Klassen zu erben.

Base class ***Klasse*** **is inaccessible because also in** ***Klasse***
[Auf die Basisklasse *Klasse* kann nicht zugegriffen werden, weil sie sich auch in *Klasse* befindet]
(Compiler, Warnung) Die Verwendung einer Klasse sowohl als direkte als auch als indirekte Basisklasse ist nicht zulässig, da die Elemente automatisch mehrdeutig werden. Definieren Sie die Basisklasse an beiden Positionen als virtuell.

Base class ***Klasse*** **is included more than once**
[Basisklasse *Klasse* mehr als einmal vorhanden]
(Compiler, Fehler) Eine C++-Klasse kann von beliebig vielen Basisklassen abgeleitet werden. Sie kann jedoch nur einmal direkt von einer gegebenen Klasse abgeleitet werden.

Base class *Klasse* **is initialized more than once**
[Basisklasse *Klasse* mehr als einmal initialisiert]
(Compiler, Fehler) Im Konstruktor einer C++-Klasse enthält die Liste der Initialisierungen nach dem Konstruktorkopf mehr als einmal die Basisklasse *Klasse*.

Base initialization without a class name is now obsolete
[Initialisierung der Basisklasse ohne Namen ist überholt]
(Compiler, Fehler) Frühere Versionen von C++ initialisierten eine Basisklasse mit der Parameterliste des Konstruktors der Basisklasse. Jetzt muß der Klassenname angegeben werden, was den Quelltext lesbarer macht und bei mehrfachen Basisklassen auch nötig ist. Früher:

```
abgeleitet::abgeleitet(int i) : (i, 10) {...}
```

Jetzt:

```
abgeleitet::abgeleitet(int i) : basis(i, 10) {...}
```

Bit field cannot be static
[Bitfeld darf nicht statisch sein]
(Compiler, Fehler) Nur Datenelemente einer gewöhnlichen C++-Klasse können **static** deklariert werden, Bitfelder nicht.

Bit field too large
[Bitfeld zu groß]
(Compiler, Fehler) Den Fehler erhalten Sie, wenn ein Bitfeld größer als 16 Bit ist.

Bit fields must be signed or unsigned int
[Bitfelder müssen vom Typ **signed** oder unsigned **int** sein]
(Compiler, Fehler) In ANSI C müssen Bitfelder als Integertyp **signed** oder **unsigned int** deklariert sein (und beispielsweise nicht als **char** oder **long**).

Bit fields must be signed or unsigned int
[Bitfelder müssen vom Typ **signed int** oder **unsigned int** sein]
(Compiler, Warnung) In ANSI C können Bitfelder nicht vom Typ **signed char** oder **unsigned char** sein. Wenn nicht im strengen ANSI-Modus compiliert wird, läßt der Compiler diese Konstrukte zu, gibt jedoch eine Warnung aus.

Bit fields must contain at least one bit
[Bitfelder müssen mindestens ein Bit enthalten]
(Compiler, Fehler) Ein benanntes Bitfeld darf nicht 0 Bit (oder weniger als 0) enthalten. Ein nicht benanntes Bitfeld darf 0 Bit enthalten, um das folgende Bitfeld an einer Byte-Grenze auszurichten (oder IDE-Option *Compiler/Code-Generierung/Ausrichtung an geraden Adressen*), an einer WORD-Grenze. In C++ müssen Bitfelder Integertypen sein; dazu gehören auch **enum**-Typen.

Bit fields must have integral type
[Bitfelder müssen Integertypen sein.]
(Compiler, Fehler) In C++ müssen Bitfelder ein Integertyp sein; dazu gehören auch **enum**-Typen.

Body has already been defined for function ***Funktion***
[Funktionsrumpf für diese Funktion schon definiert]
(Compiler, Fehler) Für eine Funktion dieses Namens und Typs existiert schon ein Funktionsrumpf. Es darf nur ein Funktionsrumpf vorhanden sein.

Both return and return of a value used
[Funktion endet an einer Stelle mit return, an anderer Stelle mit return (*Wert*)]
(Compiler, Warnung) Die aktuelle Funktion hat **return**-Anweisungen mit und ohne Werte. Das ist in C legal, ergibt aber meistens einen Fehler. Eventuell wurde eine **return**-Anweisung am Ende einer Funktion vergessen.

Call of nonfunction
[Angegebener Bezeichner ist kein Funktionsname]
(Compiler, Fehler) Bei dem für den Aufruf verwendeten Namen handelt es sich nicht um einen Funktionsnamen. Dieser Fehler tritt normalerweise dann auf, wenn die Funktion fehlerhaft deklariert wurde oder wenn Sie bei der Angabe des Namens einen Tippfehler gemacht haben.

Call to function ***Funktion*** **with no prototype**
[Aufruf der Funktion *Funktion* ohne vorherige Definition eines Prototyps]
(Compiler, Warnung) Diese Warnung erhalten Sie nach Aktivierung von *Prototypes required*, wenn Sie die Funktion *Funktion* aufrufen, ohne zuvor einen Prototyp für diese Funktion anzugeben.

Cannot add or subtract relocatable symbols
[Relozierbare Symbole können nicht addiert oder subtrahiert werden]
(Compiler, Fehler) Die einzige arithmetische Operation, die für ein relozierbares Symbol in einem Assembler-Operanden ausgeführt werden kann, ist die Addition oder Subtraktion einer Konstante. Variablen, Prozeduren, Funktionen und Labels sind relozierbare Symbole. Wenn *Var* eine Variable und *Const* eine Konstante ist, dann sind die beiden Befehle

```
MOV AX,Const+Const
```

```
MOV AX,Var+Const
```

gültig,

```
MOV AX,Var+Var
```

dagegen nicht.

Cannot allocate a reference
[Für eine Referenz kann kein Speicherplatz reserviert werden]
(Compiler, Fehler) Es wurde versucht, mit dem Operator **new** eine Referenz zu erzeugen. Das ist jedoch unzulässig, weil Referenzen keine Objekte sind und deshalb nicht mit Hilfe von **new** erzeugt werden können.

Bezeichner **cannot be declared in an anonymous union**
[*Bezeichner* kann nicht als anonyme Variante deklariert werden]
(Compiler, Fehler) Der Compiler fand eine Deklaration einer Elementfunktion

oder eines statischen Elements in einer anonymen Variante. Derartige Varianten können nur Datenelemente enthalten.

Funktion1* cannot be distinguished from *Funktion2
[*Funktion1* nicht von *Funktion2* unterscheidbar]
(Compiler, Fehler) Die Liste der Parametertypen in den Deklarationen dieser beiden Funktionen ist nicht verschieden genug, um die Funktionen unterscheiden zu können. Verändern Sie die Reihenfolge der Parameter oder ändern Sie den Typ eines Parameters in einer der Listen.

Cannot call near class member function with a pointer of type *Typ*
[Eine Elementfunktion einer **near**-Klasse kann nicht mit einem Zeiger vom Typ *Typ* aufgerufen werden]
(Compiler, Fehler) Elementfunktionen von **near**-Klassen können nicht mit Elementzeigern vom Typ **far** oder **huge** aufgrufen werden. (Zur Erinnerung: Klassen sind in den Speichermodellen *Tiny*, *Small* und *Medium* per Voreinstellung vom Typ **near**.) Dies trifft auch auf Aufrufe mit Zeigern auf Elementfunktionen zu. Ändern Sie entweder den Zeiger auf den Typ **near** um oder deklarieren Sie die Klasse als **far**.

Cannot cast from *Typ1* to *Typ2*
[Typenumwandlung von *Typ1* nach *Typ2* nicht zulässig]
(Compiler, Fehler) Eine Umwandlung von *Typ1* in Typ *Typ2* ist nicht zulässig. In C kann ein Zeiger in einen anderen Zeiger oder einen Integertyp umgewandelt werden. Ein Integertyp kann in jeden Integer-, Gleitkomma- oder Zeigertyp umgewandelt werden, ein Gleitkommatyp in einen Gleitkomma- oder Integertyp. Strukturen und Arrays können nicht ineinander oder in andere Typen umgewandelt werden. Der Datentyp **void** kann ebensowenig umgewandelt werden.

In C++werden benutzerdefinierte Umwandlungsvorschriften und Konstruktoren geprüft. Sind keine benutzerdefinierten Regeln vorhanden, gelten die oben genannten Regeln (außer für Zeiger auf Klassenelemente). Bei Integertypen darf nur die Konstante Null in den Zeiger auf ein Element umgewandelt werden. Ein Elementzeiger kann in einen Integertyp oder in einen ähnlichen Elementzeiger umgewandelt werden. Ein ähnlicher Zeiger ist ein Zeiger auf ein Datenelement, falls der erste Zeiger dorthin zeigt, oder auf eine Elementfunktion, falls der erste Zeiger auf eine solche zeigt.

Die qualifizierende Klasse des umzuwandelnden Typs muß dieselbe oder eine Basisklasse des ursprünglichen Typs sein.

Cannot convert *Typ1* to *Typ2*
[*Typ1* kann nicht in *Typ2* konvertiert werden]
(Compiler, Fehler) Eine Zuweisumg, eine Initialisierung oder ein Ausdruck erfordert, daß die angegebene Typumwandlung durchgeführt wird. Die Umwandlung ist aber nicht zulässig.

Cannot create instance of abstract class *Klasse*
[Instanz der abstrakten Klasse *Klasse* kann nicht erzeugt werden]
(Compiler, Fehler) Abstrakte Klassen (das heißt Klassen mit abstrakten Funktionen) können nicht direkt, sondern nur über ihre Ableitungen verwendet werden.

Cannot define a pointer or reference to a reference
[Definition eines Zeigers oder einer Referenz auf eine Referenz nicht zulässig]
(Compiler, Fehler) Ein Zeiger oder eine Referenz auf eine Referenz ist nicht zulässig.

Cannot find *Klasse::Klasse(Klasse&)* **to copy a vector**
[Kann Vektor nicht kopieren, Referenz auf *Klasse* fehlt]
(Compiler, Fehler) Enthält eine C++-Klasse *Klasse1* einen Vektor (Array) aus **class** *Klasse2*, und Sie wollen aus einem Objekt der *Klasse1* ein anderes Objekt der *Klasse1* konstruieren, muß es auch einen Konstruktor *Klasse2::Klasse2(Klasse2&)* geben, um die Elemente des Vektors konstruieren zu können. Dieser Konstruktor braucht nur ein Argument, die Referenz auf seine Klasse.

Dieser Kopierkonstruktor wird üblicherweise vom Compiler bereitgestellt. Wenn Sie aber einen Konstruktor für die *Klasse2* mit einem Parameter des Typs *Klasse2&* und zusätzlichen Parametern mit Vorgabewerten definiert haben, kann kein Kopierkonstruktor vom Compiler erzeugt werden. Der Grund dafür ist, daß *Klasse2::Klasse(Klasse2&)* und *Klasse2::Klasse2(Klasse2&, int = 1)* nicht unterscheidbar sind. Sie müssen den Konstruktor erneut definieren, so daß nicht alle Parameter Vorgabewerte haben. Dann können Sie einen Kopierkonstruktor definieren oder vom Compiler erzeugen lassen.

Cannot find *Klasse::***Operator=(** *Klasse&***) to copy a vector**
[Zum Kopieren eines Array kann kein Operator *Klasse::operator=(Klasse&)* gefunden werden]
(Compiler, Fehler) Enthält eine C++ Klasse *Klasse1* einen Vektor (Array) aus Klasse *Klasse2*, und wollen Sie ein Objekt des Typs *Klasse1* kopieren, muß ein Zuweisungsoperator *Klasse2::operator=(Klasse2&)* vorhanden sein, um die Elemente des Vektors zu kopieren. Normalerweise wird ein derartiger Operator vom Compiler automatisch erzeugt. Wenn Sie einen *operator=* für die *Klasse2* definiert haben, der keine Parameter vom Typ *Klasse2&* akzeptiert, wird vom Compiler nicht automatisch ein Zuweisungsoperator geliefert – Sie müssen das tun.

Cannot find default constructor to initialize array element of type *Klasse*
[Kein Standard-Konstruktor für die Initialisierung von Array-Elementen von Typ *Klasse* vorhanden]
(Compiler, Fehler) Wenn Sie ein Array einer Klasse deklarieren, die über Konstruktoren verfügt, müssen Sie entweder jedes Array-Element explizit initialisieren oder für die Klasse muß ein Standardkonstruktor definiert sein, mit dessen Hilfe die Array-Elemente initialisiert werden, die keinen expliziten Initialisierer besitzen. Der Compiler definiert einen Standardkonstruktor für eine Klasse – außer Sie haben selbst Konstruktoren für die Klasse definiert.

Cannot find default constructor to initialize base class ***Klasse***
[Kein Standardkonstruktor für die Initialisierung der Basisklasse *Klasse* vorhanden]
(Compiler, Fehler) Wenn eine abgeleitete C++ Klasse *Klasse2* initialisiert wird, muß jede Basisklasse *Klasse1* zuerst initialisiert werden. Wenn der Konstruktor der *Klasse2* keinen Konstruktor für *Klasse1* bestimmt (als Bestandteil des Kopfes von *Klasse2*), muß es einen Konstruktor *Klasse1::Klasse1*() für die Basisklasse geben. Dieser Konstruktor ohne Parameter wird Standardkonstruktor genannt. Der Compiler liefert automatisch einen Standardkonstruktor, außer wenn Sie bereits einen Konstruktor für die Klasse *Klasse1* definiert haben. In diesem Fall stellt der Compiler den Standardkonstruktor nicht zur Verfügung, Sie müssen selbst einen Konstruktor bereitstellen.

Cannot find default ***constructor*** **to initialize member** ***Bezeichner***
[Zur Initialisierung des Elements *Bezeichner* kann kein Standardkonstruktor gefunden werden]
(Compiler, Fehler) Wenn eine C++Klasse *Klasse1* ein Element der Klasse *Klasse2* enthält, und Sie ein Objekt der *Klasse1* nicht aus einem anderen Objekt der *Klasse1* erstellen wollen, muß ein Konstruktor *Klasse2::Klasse2*() existieren, damit das Element erzeugt werden kann. Dieser Konstruktor ohne Parameter wird Standardkonstruktor genannt. Der Compiler liefert automatisch einen Standardkonstruktor, außer wenn Sie bereits einen Konstruktor für die Klasse *Klasse2* definiert haben. In diesem Fall stellt der Compiler den Standardkonstruktor nicht zur Verfügung, Sie müssen selbst einen Konstruktor bereitstellen.

Cannot generate ***Funktion*** **from template function** ***Schablone***
[*Funktion* kann nicht aus der Template-Funktion *Schablone* generiert werden]
(Compiler, Fehler) Eine Template-Funktion wird aufgerufen, die Funktion kann aber aus dem Template *Schablone* nicht erzeugt werden.

Cannot have a near class member in a far class
[In einer **far**-Klasse darf kein **near**-Element vorhanden sein]
(Compiler, Fehler) Alle Elemente einer C++**far**-Klasse müssen vom Typ **far** sein. Dieses Element ist als **near** in einer **far**-Klasse deklariert.

Cannot have a non-inline function in a local class
[Nicht-Inline-Funktionen sind in einer lokalen Klasse nicht erlaubt]
(Compiler, Fehler) Alle Elemente von Klassen, die als lokale zu einer Funktion deklariert sind, müssen in der Klassendefinition vollständig definiert sein. Das bedeutet, daß lokale Klassen keine statischen Datenelemente enthalten können und daß die Rümpfe aller Elementfunktionen dieser Klassen in der Klassendefinition definiert sein müssen.

Cannot have a static data member in a local class
[Statische Daten sind in einer lokalen Klasse nicht erlaubt]
(Compiler, Fehler) Alle Elemente von Klassen, die als lokale zu einer Funktion

deklariert sind, müssen in der Klassendefinition vollständig definiert sein. Das bedeutet, daß lokale Klassen keine statischen Datenelemente enthalten können und daß die Rümpfe aller Elementfunktionen dieser Klassen in der Klassendefinition definiert sein müssen.

Cannot initialize a class member here
[Klassenelement kann hier nicht initialisiert werden]
(Compiler, Fehler) Einzelne Elemente einer Struktur, Variante oder einer Klasse (in C++) dürfen keine Initialisierer haben. Strukturen oder Varianten können als Ganzes mit Initialisierern in geschweiften Klammern initialisiert werden. Eine Klasse (in C++) kann nur über einen Konstruktor initialisiert werden.

Cannot initialize *Typ1* with *Typ2*
[Ein Objekt *Typ1* kann nicht mit einem Wert *Typ2* initialisiert werden]
(Compiler, Fehler) Sie haben versucht, ein Objekt von Typ *Typ1* mit einem Wert von Typ *Typ2* zu initialisieren. Das ist jedoch nicht erlaubt. Für Initialisierungen gelten grundsätzlich die normalen Zuweisungsregeln.

Cannot modify a const object
[**const**-Objekt kann nicht geändert werden]
(Compiler, Fehler) Diese Meldung signalisiert, daß für ein als **const** deklariertes Objekt eine unzulässige Operation durchgeführt wurde, wie z.B. eine Zuweisung an das Objekt.

Cannot overload 'main'
[*main* kann nicht überladen werden]
(Compiler, Fehler) *main* ist die einzige Funktion, die nicht überladen werden kann.

***Funktion* cannot return a value**
[*Funktion* kann keinen Wert zurückliefern]
(Compiler, Fehler) Eine Funktion mit einem Rückgabewert vom Typ **void** enthält eine **return**-Anweisung, die z.B. einen **int**-Wert zurückliefert.

***Bezeichner* cannot start an argument declaration**
[Am Beginn einer Argument-Deklaration darf kein *Bezeichner* stehen]
(Compiler, Fehler) In einer Funktionsdeklaration wurde ein undefinierter *Bezeichner* am Beginn eines Arguments gefunden. Häufig weist der Typname einen Tippfehler auf oder es fehlt die Typdeklaration (durch Fehlen der entsprechenden Header-Datei).

Cannot write GRPDEF list, extended dictionary aborted
[GRPDEF-Liste kann nicht geschrieben werden, das erweiterte Verzeichnis wird nicht angelegt]
(Bibliotheksverwaltung, Fehler) Die Bibliotheksverwaltung kann das erweiterte Verzeichnis nicht an des Ende der Bibliotheksdatei schreiben. Normalerweise ist in diesem Fall nicht genügend Speicherplatz auf der Platte vorhanden.

Case bypasses initialization of a local variable
[**Case** umgeht die Initialisierung einer lokalen Variable]
(Compiler, Fehler) In C++ ist es nicht erlaubt, die Initialisierung einer lokalen Variable zu umgehen. In diesem Fall gibt es ein **case**-Label, das die Steuerung hinter diese lokale Variable transferieren kann.

Case outside of switch
[**case**-Marke außerhalb von **switch**]
(Compiler, Fehler) Der Compiler hat eine **case**-Anweisung angetroffen, die außerhalb einer **switch**-Anweisung steht. Der Grund dafür liegt häufig in einer fehlenden oder falsch plazierten geschweiften Klammer.

Case statement missing :
[**case**-Anweisung ohne Doppelpunkt]
(Compiler, Fehler) Auf das reservierte Wort **case** muß ein Konstantenausdruck und danach ein Doppelpunkt folgen. In der **case**-Anweisung fehlt entweder der Doppelpunkt im Ausdruck oder es steht ein zusätzliches Zeichen vor dem Doppelpunkt.

Character constant must be one or two characters long
[Zeichenkonstante zu lang]
(Compiler, Fehler) Zeichenkonstanten dürfen nur aus einem oder zwei Zeichen bestehen.

Class *Klasse* may not contain pure functions
[Die Klasse *Klasse* darf keine abstrakten Funktionen enthalten]
(Compiler, Fehler) Die zu deklarierende Klasse kann nicht abstrakt sein und darf deshalb keine abstrakten Funktionen enthalten.

Class member *Element* declared outside its class
[Klassenelement *Element* außerhalb seiner Klasse deklariert]
(Compiler, Fehler) Elementfunktionen einer C++ Klasse können nur in der Klassendeklaration deklariert werden. Sie können nicht wie andere Funktionen mehrfach oder an anderen Stellen deklariert werden.

Code has no effect
[Erzeugter Code ist wirkungslos]
(Compiler, Warnung) Diese Warnung erhalten Sie, wenn der Compiler auf eine Anweisung mit Operatoren trifft, die keine Auswirkung hat. So hat beispielsweise folgende Anweisung

```
a + b;
```

keine Auswirkung auf die beiden Variablen. Die Operation ist nicht erforderlich und weist wahrscheinlich auf einen Fehler hin.

Common segment exceeds 64K
[Gemeinsames Segment ist größer als 64 KByte]
(Linker, Fehler) Das Programm hat mehr als 64 KByte uninitialisierte **near**-

Daten. Versuchen Sie in diesem Fall, einige uninitialisierte Daten als **far** zu deklarieren.

Compiler could not generate copy constructor for class *Klasse*
[Compiler konnte den Kopierkonstruktor für die Klasse *Klasse* nicht erzeugen]
(Compiler, Fehler) Der Compiler konnte den erforderlichen Kopierkonstruktor nicht den Sprachregeln gemäß erstellen.

Compiler could not generate default constructor for class *Klasse*
[Compiler konnte den Standardkonstruktor für die Klasse *Klasse* nicht erzeugen]
(Compiler, Fehler) Der Compiler konnte den erforderlichen Standardkonstruktor nicht den Sprachregeln gemäß erstellen.

Compiler could not generate operator= for class *Klasse*
[Compiler konnte operator= für die Klasse *Klasse* nicht erzeugen]
(Compiler, Fehler) Der Compiler konnte den erforderlichen Zuweisungsoperator nicht den Sprachregeln gemäß erstellen.

Compiler table limit exceeded
[Überlauf in einer Compilertabelle]
(Compiler, schwerer Fehler) Diese Meldung erhalten Sie, wenn es in einer internen Compilertabelle zu einem Überlauf gekommen ist. Der Grund liegt normalerweise darin, daß das zu compilierende Modul zu viele C++ Klassendeklarationen, zu viele Funktionen oder zu viele andere Einträge enthält. Mehr Hauptspeicher zur Verfügung zu stellen hilft in diesem Fall nicht; die einzige Lösung dieses Problems besteht in der Vereinfachung der zu compilierenden Datei.

Compound statement missing }
[Blockanweisung: abschließendes } fehlt]
(Compiler, Fehler) Der Compiler hat das Ende der Quelltextdatei erreicht und keine reche geschweifte Klammer gefunden; d.h. es besteht ein Klammerungsfehler.

Condition is always false
[Bedingung immer *false*]
(Compiler, Warnung) Der Compiler fand einen Wertevergleich, bei dem das Ergebnis immer *False* ist.

Condition is always true
[Bedingung immer *true*]
(Compiler, Warnung) Der Compiler fand einen Wertevergleich, bei dem das Ergebnis immer *True* ist), z.B.:

```
void proc(unsigned x)
{
   if (x >= 0)       /* immer True */
```

```
    {
    .
    .
    .
    }
}
```

Conflicting type modifiers
[Angegebene Modifizierer nicht miteinander vereinbar]
(Compiler, Fehler) Es ist eine Deklaration vorhanden, die z.B. das Schlüsselwort **far** und **near** für denselben Zeiger enthält. Es darf aber nur ein Adreß-Modifizierer für einen Zeiger und nur ein Sprach-Modifizierer (**cdecl**, **pascal** oder **interrupt**) für eine Funktion existieren.

Symbol* conflicts with module *Modul* in module *Modul
[*Symbol* ist unverträglich mit Modul *Modul* im Modul *Modul*]
(Linker, Warnung) Diese Warnung signalisiert eine Inkonsistenz in der Definition von *Symbol*. Das bedeutet, daß der Linker eine virtuelle Funktion und eine allgemeine Definition gefunden hat, die denselben Namen verwenden.

Constant expression required
[Angegebener Ausdruck ist keine Konstante]
(Compiler, Fehler) Arrays müssen mit einer konstanten Größe deklariert werden. Dieser Fehler wird häufig durch einen Tippfehler in einer #*define*-Konstante verursacht.

Constant is long
[Konstante wird als **long** gespeichert]
(Compiler, Warnung) Der Compiler ist entweder auf eine Dezimalkonstante, die größer als 32767 ist, oder auf eine Oktalkonstante, die größer als (dezimal) 65535 ist, gestoßen. Bei diesen Konstanten fehlt der explizite Zusatz von *L* oder *l*. Die Konstante wird als Konstante vom Typ **long** behandelt.

Constant member *Element* in class without constructors
[Konstantes Element *Element* in einer Klasse ohne Konstruktoren]
(Compiler, Fehler) Eine Klasse, die konstante Elemente enthält, muß mindestens einen benutzerdefinierten Konstruktor aufweisen. Andernfalls könnten diese Elemente nicht initialisiert werden.

Constant member *Element* is not initialized
[Konstantes Element *Element* nicht initialisiert]
(Compiler, Warnung) Diese C++ Klasse enthält ein konstantes Element, das nicht initialisiert wurde. Beachten Sie, daß konstante Elemente nur initialisiert werden dürfen. Ihnen darf später nichts mehr zugewiesen werden.

Constant out of range in comparison
[Vergleich verschiedener Teilbereiche]
(Compiler, Warnung) Die Quelltextdatei enthält einen Vergleich, der eine

Teilbereichskonstante betrifft, die außerhalb des vom Konstantentyp zugelassenen Bereichs liegt. So ergibt z.B. der Vergleich eines **unsigned**-Typs mit -1 keinen Sinn. Um eine **unsigned**-Konstante, die (dezimal) größer als 32767 ist, zu erhalten, können Sie für **unsigned** ein Typumwandlung durchführen (z.B. (*unsigned*)*65535*)), oder den Buchstaben *u* oder *U* an die Konstante anfügen (z.B. 65535u).

Trotz Ausgabe dieser Meldung wird der Compiler Code generieren, um den Vergleich durchzuführen. Auch wenn der Code letztendlich immer dasselbe Resultat liefert, weil z.B. ein Ausdruck vom Typ **char** mit dem Wert 4000 verglichen wird, wird der Code den Test doch durchführen.

Constant variable *Variable* **must be initialized**
[Konstante Variable *Variable* nicht initialisiert]
(Compiler, Fehler) Dieses C++-Objekt ist als **const** deklariert, wurde aber nicht initialisiert. Da ihm kein Wert zugewiesen werden kann, muß es bei der Deklaration initialisiert werden.

Constructor cannot be declared const or volatile
[Konstruktor kann nicht als **const** oder **volatile** deklariert werden]
(Compiler, Fehler) Ein Konstruktor wurde als **const** und/oder **volatile** deklariert – das ist nicht zulässig.

Constructor cannot have a return type specification
[Rückgabetyp eines Konstruktors kann nicht spezifiziert werden]
(Compiler, Fehler) Konstruktoren in C++ haben einen impliziten Rückgabetyp, der vom Compiler benutzt wird. Sie dürfen selbst keinen Rückgabetyp oder -wert deklarieren.

Conversion may lose significant digits
[Konvertierung hat eventuell den Verlust signifikanter Ziffern zur Folge]
(Compiler, Warnung) Für einen Zuweisungsoperator oder Ähnliches ist die Konvertierung vom Typ **long** oder unsigned **long** nach **int** oder unsigned **int** erforderlich.

Da Variable von Typ **int** und **long** nicht dieselbe Größe haben, könnte diese Konvertierung das Verhalten des Programms ändern, da möglicherweise signifikante Ziffern verloren gehen.

Conversion operator cannot have a return type specification
[Rückgabetyp eines Konvertierungsoperators kann nicht spezifiziert werden]
(Compiler, Fehler) Die C++-Elementfunktion für die Typumwandlung legt einen Rückgabetyp fest, der sich vom Typ selbst unterscheidet. Eine Deklaration der Umwandlungsfunktion *operator* darf keinen Rückgabetyp festlegen.

Could not find a match for *Argument*(*e*)
[Für *Argumente* keine Funktion mit passenden Parametern gefunden]
(Compiler, Fehler) Es konnte keine C++-Funktion mit Parametern gefunden werden, die zu den angegebenen Argumenten passen.

Could not find file *Dateiname*
[Datei nicht gefunden]
(Compiler, Fehler) Der Compiler kann die in der Kommandozeile angegebene Datei nicht finden.

Declaration does not specify a tag or an identifier
[Deklaration ohne »Tag« oder Bezeichner]
(Compiler, Fehler) Diese Deklaration deklariert nichts. Es kann sich z.B. um ein **struct** oder **union** ohne »Tag« oder Variable handeln. In C++ muß etwas deklariert werden.

Declaration is not allowed here
[Deklaration hier nicht erlaubt]
(Compiler, Fehler) Deklarationen dürfen nicht als Steueranweisungen für **while**, **for**, **do**, **if** oder **switch** benutzt werden.

Declaration missing ;
[Es fehlt ein Strichpunkt in der Deklaration]
(Compiler, Fehler) Ihre Quelltextdatei enthält eine Deklaration ohne abschließenden Strichpunkt.

Declaration syntax error
[Ungültige Syntax in einer Deklaration]
(Compiler, Fehler) Ihre Quelltextdatei enthält eine Deklaration, in der ein Symbol fehlt oder die überzählige Symbole aufweist.

Declaration terminated incorrectly
[Ende der Deklaration nicht korrekt]
(Compiler, Fehler) Nach der Deklaration steht ein zusätzliches oder falsches Zeichen, wie z.B. ein Semikolon nach einem Funktionsrumpf. Bei einer C++ Elementfunktion, die in einer Klasse zwischen Kopf und der ersten linken Klammer deklariert wird, taucht dieser Fehler ebenfalls auf.

Declaration was expected
[Deklaration erwartet]
(Compiler, Fehler) Hier wurde eine Deklaration erwartet, aber nicht gefunden. Dies ist meist mit einem fehlenden Begrenzer wie Komma, Strichpunkt, rechte Klammer usw. zu erklären.

Declare operator delete (void*) or (void*, size_t)
[Operator **delete** muß als (*void**) oder (*void**, *size_t*) deklariert werden]
(Compiler, Fehler) Korrekt deklarieren Sie **delete** mit einem einzelnen void***-**Parameter oder einem zweiten zusätzlichen Parameter des Typs *size_t*. Die zweite Version hat Vorrang vor der ersten. Der globale Operator **delete** kann nur mit einem einzelnen Parameter deklariert werden.

Declare type *Typ* prior to use in prototype
[*Typ* muß vor Verwendung in Prototyp deklariert werden]
(Compiler, Warnung) Wenn sich ein Funktionsprototyp auf einen Strukturtyp bezieht, der nicht vorher deklariert wurde, ist die Deklaration innerhalb und außerhalb des Prototypen nicht identisch. Beispiel:

```
int func(struct s *ps);
struct s{ /*....*/}
```

Da die Struktur *s* nicht im Gültigkeitsbereich des Prototypen von *func* liegt, ist der Zeiger *ps* Zeiger auf ein nicht definiertes *struct s*, und nicht derselbe Typ wie das später deklarierte *struct s*. Später würden Warnungen und Fehlermeldungen über unkompatible Typen auftreten, die ohne diese Warnung kaum erklärbar wären. Zur Lösung des Problems sollte die Deklaration von *struct s* vor jedem Prototyp stehen, der sich auf sie bezieht. Es kann auch die unvollständige Typdeklaration *struct s* vor jeden Prototyp gestellt werden, der sich auf sie bezieht.

Ist der Funktionsparameter ein **struct** anstelle eines Zeigers auf **struct**, reicht die unvollständige Deklaration nicht. In diesem Fall muß die vollständige Deklaration vor dem Prototypen stehen.

Default argument value redeclared for parameter *Parameter*
[Der voreingestelle Wert für den Parameter *Parameter* wurde neu deklariert]
(Compiler, Fehler) Wenn ein Parameter einer C++ Funktion mit einem Vorgabewert deklariert ist, kann dieser Wert nicht geändert, neu deklariert oder in einer anderen Deklaration für dieselbe Funktion weggelassen werden.

Default expression may not use local variables
[Ein Ausdruck mit Vorgabewerten kann keine lokalen Variablen verwenden]
(Compiler, Fehler) Ein Ausdruck mit voreingestellten Parametern darf keine lokalen Variablen oder anderen Parameter verwenden.

Default outside of switch
[**default** nur innerhalb von **switch** zulässig]
(Compiler, Fehler) Der Compiler trifft auf eine **default**-Anweisung außerhalb einer **switch**-Anweisung. Hier handelt es sich meistens um eine fehlende oder falsch plazierte geschweifte Klammer.

Default value missing
[Vorgabewert fehlt]
(Compiler, Fehler) Deklariert eine C++ Funktion einen Parameter mit Vorgabewert, müssen alle folgenden Parameter auch einen solchen haben. In dieser Deklaration war dies nicht der Fall.

Default value missing following parameter *Parameter*
[Nach dem Parameter *Parameter* fehlt der Vorgabewert]
(Compiler, Fehler) Alle Parameter, die nach dem ersten Parameter mit einem Vorgabewert folgen, müssen ebenfalls Vorgabewerte erhalten.

Define directive needs an identifier
[#*define*-Direktive benötigt einen Bezeichner]
(Compiler, Fehler) Direkt nach #*define* muß als erstes der Name eines Bezeichners folgen. Der Compiler hat aber ein anderes (Nicht-Whitespace-)Zeichen angetroffen.

Symbol* defined in module *Modul* is duplicated in module *Modul
[Im Modul *Modul* definiertes Symbol ist im Modul *Modul* noch einmal vorhanden]
(Linker, Fehler oder Warnung) Sie erhalten diese Meldung, wenn es zwischen zwei Symbolen (öffentlichen oder gemeinsamen) zu Problemen kommt. Normalerweise tritt diese Meldung dann auf, wenn ein Symbol in zwei Modulen definiert wurde. Ein Fehler resultiert daraus, wenn die beiden identischen Symbole in der (oder den) .OBJ-Datei(en) stehen. Der Linker kann dann nicht entscheiden, welches Symbol gültig ist. Eine Warnung erhalten Sie, wenn der Linker ein Symbol in einer Bibliothek und das Duplikat in einer .OBJ-Datei findet; in diesem Fall verwendet der Linker das Symbol in der .OBJ-Datei.

Delete array size missing]
[] fehlt bei einer Array-Größenangabe eines *delete*-Operators]
(Compiler, Fehler) Dem Array-Spezifizierer in einem *delete*-Operator fehlt die rechte eckige Klammer.

Destructor cannot be declared const or volatile
[Ein Destruktor kann nicht vom Typ **const** oder **volatile** sein]
(Compiler, Fehler) Ein Destruktor wurde als **const** oder **volatile** deklariert – das ist nicht zulässig.

Destructor cannot have a return type specification
[Destruktor darf keine Festlegung eines Rückgabetyps enthalten]
(Compiler, Fehler) Die Festlegung eines Rückgabetyps für einen Destruktor ist nicht zulässig.

Destructor for *Klasse* is not accessible
[Destruktor für *Klasse* nicht zugänglich]
(Compiler, Fehler) Der Destruktor für diese C++ Klasse ist vom Typ **protected** oder **private** und kann hier nicht verwendet werden. Ist ein Klassendestruktor **private**, kann die Klasse nicht gelöscht und daher auch nie benutzt werden – vermutlich ein Programmierfehler.

Auf einen Destruktor von Typ **protected** kann nur über abgeleitete Klassen zugegriffen werden. Dies ist eine praktische Methode, um keine Instanzen einer Basisklasse, sondern nur von abgeleiteten Klassen zuzulassen.

Destructor for *Klasse* required in conditional expression
[Destruktor für *Klasse* im bedingten Ausdruck erforderlich]
(Compiler, Fehler) Wenn der Compiler eine temporäre lokale Variable in einem bedingten Ausdruck erzeugen muß, gibt es keine günstige Stelle für den Aufruf des Destruktors, denn die Variable kann, muß aber nicht, initialisiert sein. Die

temporäre Variable kann explizit erzeugt werden (mit *Klassenname* (*val*,*val*)) oder implizit mit anderem Programmcode. Entfernen Sie den temporären Wert aus Ihrem Programm.

Destructor name must match the class name
[Name des Destruktors muß zur Klasse passen]
(Compiler, Fehler) In einer C++Klasse steht die Tilde (~) vor der Deklaration des Klassendestruktors, der genauso wie die Klasse heißen muß. Hier steht die Tilde vor einem anderen Namen.

Divide error
[Divisionsfehler]
(Laufzeit, Fehler) Diese Meldung erscheint bei einer Integerdivision durch 0, z.B.:

```
int n = 0;
n = 2 / n;
```

Der Fehler kann von der Funktion *signal* abgefangen werden. Wenn nicht, wird *abort* aufgerufen, um das Programm zu beenden.

Division by zero
[Division durch Null]
(Compiler, Fehler) Ein Konstantenausdruck in Ihrer Quelltextdatei enthält einen Divisions- oder Modulo-Operator und den Divisor Null.

Division by zero
[Division durch Null]
(Compiler, Warnung) Ein Ausdruck mit Divisions- oder Modulo-Operator hat eine literale Null (die Zahl 0) als Divisor.

do statement must have a while
[Schleife mit **do** muß **while** besitzen]
(Compiler, Fehler) Ihr Quelltext enthält eine *do*-Anweisung ohne das abschließende Schlüsselwort **while**.

DOS error, ax= *Nummer*
[DOS-Fehler, Fehlercode =*Nummer*]
(Linker, schwerer Fehler) Dieser Fehler tritt auf, wenn ein DOS-Aufruf einen »unerwarteten« Fehler liefert. Der ausgegebene *ax*-Wert ist der Fehlercode. Die Meldung deutet auf einen internen Linker- oder DOS-Fehler hin. Die einzigen vom Linker ausgeführten DOS-Aufrufe, bei denen dieser Fehler auftreten kann, sind *read*, *write*, *seek* und *close*.

do-while statement missing (
[*do-while*-Anweisung ohne (]
(Compiler, Fehler) Linke (fehlt in einer *do*-Anweisung hinter dem Schlüsselwort **while**.

do-while statement missing)
[**do-while**-Anweisung ohne)]
(Compiler, Fehler) Rechte) fehlt in einer *do*-Anweisung hinter dem Ausdruck mit der Abbruchbedingung der Schleife.

do-while statement missing ;
[**do-while**-Anweisung ohne Strichpunkt]
(Compiler, Fehler) Der Strichpunkt hinter der rechten Klammer) fehlt in einer *do*-Anweisung.

Duplicate case
[Mehrere **case**-Marken mit derselben Konstanten]
(Compiler, Fehler) Jede **case**- in einer **switch**-Anweisung muß einen eindeutigen, konstanten Ausdruckswert haben.

***Dateiname* (*Zeilennummer*): Duplicate external name in exports**
[Doppelter externer Name im EXPORTS-Bereich]
(Linker, Warnung) Zwei Exportfunktionen im EXPORTS-Bereich einer Moduldefinitionsdatei haben denselben externen Namen, z. B.:

```
EXPORTS
  AnyProc1=MyProc
  AnyProc2=MyProc
```

***Dateiname* (*Zeilennummer*): Duplicate internal name in exports**
[Doppelter interner Name im EXPORTS-Bereich]
(Linker, Warnung) Zwei Exportfunktionen im EXPORTS-Bereich einer Moduldefinitionsdatei haben denselben internen Namen, z.B.:

```
EXPORTS
  AnyProc=MyProc1
  AnyProc=MyProc2
```

***Dateiname* (*Zeilennummer*): Duplicate internal name in imports**
[Doppelter interner Name im IMPORTS-Bereich]
(Linker, Warnung) Zwei Importfunktionen im IMPORTS-Bereich einer Moduldefinitionsdatei definieren den gleichen internen Namen; zum Beispiel:

```
IMPORTS
  AnyProc=MyMod1.MyProc1
  AnyProc=MyMod2.MyProc2
```

oder

```
IMPORTS
  MyMod1.MyProc
  MyMod2.MyProc
```

Duplicate ordinal *Nummer* in exports
[Ordinalwert *Nummer* tritt in Exporten doppelt auf]
(Linker, Warnung) Sie erhalten diese Warnung, wenn der Linker zwei Exporte

mit demselben Ordinalwert findet. Überprüfen Sie zuerst die Moduldefinitionsdatei, um sicherzustellen, daß im EXPORTS-Bereich keine identischen Ordinalwerte stehen. Wenn das nicht der Fall ist, ist beim Linken von Modulen, die Exporte über ihren Ordinalwert spezifizieren, folgendes geschehen: entweder geben zwei Export-Records denselben Ordinalwert an oder der EXPORTS-Bereich in der Moduldefinitionsdatei enthält einen bereits in einem Export-Record vorhandenen Ordinalwert.

Export-Records (EXPDEF) sind Kommentar-Records in .OBJ-Dateien oder Bibliotheken, die bestimmte Variablen als zu exportierende angeben. Optional können diese Records Ordinalwerte angeben, wenn über Ordinalwerte (anstatt über den Namen) exportiert wird.

Enum syntax error
[**enum**-Deklaration mit ungültiger Syntax]
(Compiler, Fehler) Eine **enum**-Deklaration enthält eine fehlerhafte Bezeichnerliste.

Error directive: ***Meldung***
[#*error*-Anweisung: *Meldung*]
(Compiler, schwerer Fehler) Der Text der in der Quelltextdatei bearbeiteten #*error*-Anweisung wird ausgegeben.

Error writing output file
[Fehler beim Schreiben der Ausgabedatei]
(Compiler, schwerer Fehler) Ein DOS-Fehler hindert Turbo C++ daran, eine .OBJ-, .EXE- oder temporäre Datei zu schreiben. Überprüfen Sie die Option *Optionen/Verzeichnisse.../Ausgabeverzeichnis* und stellen Sie sicher, daß das angegebene Verzeichnis gültig ist. Überprüfen Sie auch, ob genügend Speicherplatz zur Verfügung steht.

Expression expected
[Ausdruck erwartet]
(Compiler, Fehler) Hier wird ein Ausdruck erwartet, der aber nicht mit dem aktuellen Symbol beginnen kann. Dies tritt eventuell beim Steuerausdruck für eine **if**- oder **while**-Klausel oder beim Initialisieren einer Variablen auf, oft wegen eines versehentlich eingefügten oder gelöschten Zeichens.

Expression of scalar type expected
[Skalarer Ausdruck erwartet]
(Compiler, Fehler) Die Operatoren *Nicht* (**!**), *Inkrement* (**++**) und *Dekrement* (**--**) erfordern einen skalaren Ausdruck. Erlaubte Typen sind **char**, **short**, **int**, **long**, **enum**, **float**, **double**, **long double** und Zeigertypen.

Expression syntax
[Syntaxfehler im Ausdruck]
(Compiler, Fehler) Diese Meldung steht für alle unspezifischen Fehler in Ausdrücken, wie z.B. aufeinanderfolgende Operatoren, fehlende oder falsche Klammern oder ein fehlender Strichpunkt in der vorausgehenden Anweisung.

Grund extended dictionary not created
[Erweitertes Verzeichnis nicht anlegt]
(Bibliotheksverwaltung, Warnung) Die Bibliotheksverwaltung konnte das erweiterte Verzeichnis wegen *Grund* nicht anlegen.

extern variable cannot be initialized
[Externe Variable kann nicht initialisiert werden]
(Compiler, Fehler) Die Speicherklasse **extern** bedeutet für eine Variable, daß sie hier deklariert, aber nicht definiert und kein Speicherplatz für sie reserviert wird. Deshalb kann sie in der Deklaration noch nicht initialisiert werden.

Extra argument in template class name ***Schablone***
[Überzähliges Argument im Namen der Template-Klasse *Schablone*).
(Compiler, Fehler) Der Name einer Template-Klasse enthält zu viele Werte für seine formalen Parameter.

Extra parameter in call
[Überzählige Argumente bei Funktionsaufruf]
(Compiler, Fehler) Ein Funktionsaufruf über einen mit einem Prototyp definierten Zeiger hat zu viele Argumente.

Extra parameter in call to ***Funktion***
[Überzähliger Parameter bei Aufruf der Funktion *Funktion*]
(Compiler, Fehler) Der Aufruf der angegebenen Funktion (die mit einem Prototyp definiert ist) hat zu viele Argumente.

File must contain at least one external declaration
[Datei muß mindestens eine externe Deklaration enthalten]
(Compiler, Fehler) Der Inhalt dieser Compilierungseinheit war logisch leer, weil sie keine externe Deklaration enthielt. In ANSI C und C++ muß etwas in der zu compilierenden Einheit deklariert werden.

File name too long
[Dateiname zu lang]
(Compiler, Fehler) Der Dateiname in der #*include*-Anweisung ist für den Compiler zu lang. Pfadnamen in DOS dürfen nicht mehr als 79 Zeichen umfassen.

Dateiname **file not found**
[*Dateiname* nicht gefunden]
(Bibliotheksverwaltung, Fehler) Die IDE erzeugt die Bibliothek, indem zuerst die vorhandene Bibliothek entfernt und danach neu aufgebaut wird. Wenn Objekte nicht existieren, wird die Bibliothek als unvollständig betrachtet und diese Fehlermeldung ausgegeben. Wenn die IDE meldet, daß ein Objekt nicht existiert, wurde entweder das Quellmodul nicht compiliert oder es traten Fehler bei der Compilierug auf. Die Durchführung von *Compiler/Projekt aktualisieren* oder *Compiler/Alle Projekt-Dateien compilieren* sollte das Problem lösen oder melden, wo Fehler aufgetreten sind.

***Dateiname* (*Zeilennummer*): File read error**
[Fehler beim Lesen der Datei]
(Linker, schwerer Fehler) Es ist ein DOS-Fehler aufgetreten, während der Linker die Moduldefinitionsdatei liest. Dies wird normalerweise durch ein vorzeitiges Erreichen des Dateiendes ausgelöst.

Fixup overflow at *Segment*:*xxxxh*, target = *Segment*:*xxxh* in module *Modul*
[Fixup-Überlauf bei *Segment:xxxxh*, Ziel = *Segment:xxxh* im Modul *Modul*]
(Linker, Fehler) Diese Meldung signalisiert eine falsche Code- oder Datenreferenzierung in einer .OBJ-Datei, die der Linker zur Linkzeit adressiert.

Sie tritt sehr häufig bei unpassenden Speichermodellen auf. Der häufigste Grund für diesen Fehler ist ein **near**-Aufruf einer Funktion, die in einem anderen Code-Segment steht. Außerdem erscheint diese Meldung, wenn Sie einen **near**-Aufruf für eine Datenvariable oder eine Datenreferenz für eine Funktion generieren. Bei dieser Fehlermeldung ist das als *Ziel* benannte Symbol die referenzierte Variable oder Funktion. Die Referenz liegt in dem angegebenen Modul. Sehen Sie sich den Quelltext dieses Moduls an, um die Referenz zu korrigieren.

In einem Assembler-Programm tritt diese Fehlerart häufig dann auf, wenn Sie eine externe Variable in einer Segmentdefinition deklariert haben, die Variable aber tatsächlich in einem anderen Segment existiert.

Bei einem Assembler- oder Hochsprachenprogramm kann es außer den eben erläuterten Fehlerquellen noch weitere Gründe für das Auftreten dieser Meldungen geben. Auch bei Turbo C++ kann diese Meldung dann auftreten, wenn Sie nicht die für ein gegebenes Speichermodell standardmäßig eingestellten Werte, sondern andere Segment- oder Gruppennamen verwenden.

Fixup overflow at *Segment*:*xxxxh*, target = *Symbol* in module *Modul*
[Fixup-Überlauf bei *Segment:xxxxh*, Ziel = *Symbol* im Modul *Modul*]
(Linker, Fehler) Diese Meldung signalisiert eine falsche Code- oder Datenreferenzierung in einer .OBJ-Datei, die der Linker zur Linkzeit adressiert. Sie tritt häufig bei unpassenden Speichermodellen auf. Der häufigste Grund für diesen Fehler ist ein **near**-Aufruf einer Funktion, die in einem anderen Code-Segment steht. Außerdem erhalten Sie diese Meldung dann, wenn Sie einen **near**-Aufruf für eine Datenvariable oder eine Datenreferenz für eine Funktion generieren. Bei dieser Fehlermeldung ist das als *Ziel* benannte Symbol die referenzierte Variable oder Funktion. Die Referenz liegt in dem angegebenen Modul. Sehen Sie sich den Quelltext dieses Moduls an, um die Referenz zu korrigieren.

In einem Assembler-Programm tritt diese Fehlerart häufig dann auf, wenn Sie eine externe Variable in einer Segmentdefinition deklariert haben, die Variable aber tatsächlich in einem anderen Segment existiert.

Bei einem Assembler- oder Hochsprachenprogramm kann es außer den eben erläuterten Fehlerquellen noch weitere Gründe für das Auftreten dieser Meldung geben. Auch bei Turbo C++ kann diese Meldung dann auftreten, wenn Sie nicht die für ein gegebenes Speichermodell standardmäßig eingestellten Werte, sondern andere Segment- oder Gruppennamen verwenden.

Floating point error: Divide by 0
[Gleitkommafehler: Division durch 0]
(Laufzeit, Fehler) Dieser schwere Fehler resultiert aus einer Gleitkommaberechnung mit einem Ergebniswert im »Unendlichen«. »Division durch 0« bedeutet, das Ergebnis ist +INF oder -INF (»infinity«), wie bei 1.0/0.0.

Floating point error: Domain
[Gleitkommafehler: Definitionsbereich]
(Laufzeit, Fehler) Dieser schwere Fehler resultiert aus einer Gleitkommaberechnung mit einem Ergebniswert im »Unendlichen«. »Definitionsbereich« bedeutet, das Ergebnis ist NAN (»not a number«, wie 0.0/0.0.

Floating point error: Overflow
[Gleitkommafehler: Überlauf]
(Laufzeit, Fehler) Dieser schwere Fehler resultiert aus einer Gleitkommaberechnung mit einem Ergebniswert im »Unendlichen«. »Überlauf« bedeutet, das Ergebnis ist +INF oder -INF (»infinity«) mit einem vollständigen Verlust der Genauigkeit, z.B. bei einer Zuweisung von 1e200 * 1e200 an einen **double**-Wert.

Floating point error: Partial loss of precision.
[Gleitkommafehler: Teilweiser Verlust der Genauigkeit]
(Laufzeit, Fehler) Diese Fehlerbedingung wird standardmäßig abgefangen und die Meldung wird nicht ausgegeben. Der teilweise Verlust der Genauigkeit wird ignoriert. Die Fehlerbedingung kann mit Hilfe von *_control87* zur Anzeige freigegeben werden.

Floating point error: Underflow.
[Gleitkommafehler: Unterlauf]
(Laufzeit, Fehler) Diese Fehlerbedingung wird standardmäßig abgefangen und die Meldung wird nicht ausgegeben. Unterlauf wird in 0.0 konvertiert. Die Fehlerbedingung kann mit Hilfe von *_control87* zur Anzeige freigegeben werden.

Floating point error: Stack fault.
[Gleitkommafehler: Stack-Fehler]
(Laufzeit, Fehler) Diese Meldung zeigt einen Überlauf des Gleitkomma-Stack an. Sie tritt normalerweise nicht auf und wird nur von Assembler-Routinen ausgelöst, die zuviele Register benutzen, kann aber auch auf eine fehlerhafte Deklaration einer Gleitkommafunktion zurückgehen.

Die Gleitkommafehler können maskiert und dadurch unterdrückt, oder mit der Funktion *signal* abgefangen werden. Einzelheiten hierzu finden Sie im *Referenzhandbuch* in der Beschreibung der Funktionen *_control87* und *signal*.

In jedem der angesprochenen Fälle gibt das Programm die Fehlermeldung aus und ruft anschließend *abort* auf, das mit der Meldung

```
Abnormal program termination
```

und anschließendem Aufruf von *_exit(3)* das Programm abbricht. Lesen Sie hierzu auch die Beschreibung von *abort* und *_exit*.

For statement missing (
[In der **for**-Anweisung fehlt (]
(Compiler, Fehler) In einer **for**-Anweisung fehlt nach dem Schlüsselwort **for** die öffnende Klammer (.

For statement missing)
[In der **for**-Anweisung fehlt))
(Compiler, Fehler) In einer **for**-Anweisung fehlt nach den Steuerausdrücken die schließende Klammer).

For statement missing ;
(In der **for**-Anweisung fehlt **;**]
(Compiler, Fehler) In einer **for**-Anweisung fehlt der Strichpunkt nach einem der Ausdrücke.

Friends must be functions or classes
[**Friends** müssen Funktionen oder Klassen sein]
(Compiler, Fehler) **Friend** einer C++Klasse muß eine Funktion oder eine andere Klasse sein.

Function call missing)
[Im Funktionsaufruf fehlt)]
(Compiler, Fehler) In der Argumentliste des Funktionsaufrufs wurde ein Syntaxfehler gefunden, z.B. eine fehlende oder falsch plazierte schließende Klammer).

Function defined inline after use as extern
[Funktion soll nach externer Verwendung als **inline** deklariert werden]
(Compiler, Fehler) Nach Verwendung einer Funktion kann diese nicht mehr **inline** deklariert werden. Verlagern Sie die Inline-Deklaration im Quelltext nach vorne oder löschen Sie die Deklaration.

Function definition cannot be a Typedef'ed declaration
[Funktionsdefinition darf keine **typedef**-Deklaration sein]
(Compiler, Fehler) In ANSI C darf ein Funktionsrumpf nicht mit **typedef** definiert sein.

Function *Funktion* cannot be static
[Die Funktion *Funktion* kann nicht **static** sein]
(Compiler, Fehler) Nur gewöhnliche Elementfunktionen und die Operatoren **new** und **delete** können als **static** deklariert werden. Konstruktoren, Destruktoren und andere Operatoren dürfen nicht **static** sein.

Function *Funktion* should have a prototype
[*Funktion* sollte einen Prototyp haben]
(Compiler, Fehler) Eine Funktion wurde ohne Prototyp im Gültigkeitsbereich aufgerufen. In C ist `int foo();` kein Prototyp; dagegen ist `int foo(int);` ein Prototyp, und somit auch in `foo(void);`. In C++ist `int foo();` ein Prototyp und dasselbe wie `foo(void);` . In C sind Prototypen für alle Funktionen sehr ratsam;

in C++ sind sie *erforderlich*. In allen Fällen dient eine Funktionsdefinition (ein Funktionskopf mit Rumpf) als Prototyp, wenn sie die erste Erwähnung dieser Funktion ist.

Function should return a value
[Funktion sollte einen Wert liefern]
(Compiler, Warnung) Die Funktion war so deklariert (vielleicht implizit), daß sie einen Wert liefern sollte. Es wurde jetzt eine **return**-Anweisung ohne Wert oder das Ende der Funktion ohne **return**-Anweisung vorgefunden. Liefern Sie einen Wert oder ändern Sie die Funktionsdeklaration in **void**.

Function should return a value
[Funktion sollte einen Wert zurückliefern]
(Compiler, Fehler) Die aktuelle Funktion soll laut Deklaration einen Wert zurückliefern, der in C++ nicht vom Typ **void** (in C nicht vom Typ **int**) ist. Der Compiler findet aber eine Rückkehr ohne Wert. In C sind **int**-Funktionen davon ausgenommen, weil ältere C-Versionen die Deklaration als **void** nicht kennen.

Functions *Funktion1* and *Funktion2* both use the same dispatch number
[*Funktion1* und *Funktion2* verwenden dieselbe Zuordnungsnummer]
(Compiler, Fehler) Problem bei der Zuordnung über DDVT.

Functions containing local destructors are not expanded inline in function *Funktion*
[Funktionen mit lokalen Destruktoren werden in der Funktion *Funktion* nicht **inline** erweitert]
(Compiler, Warnung) Sie haben eine Inline-Funktion angegeben, die Turbo C++ nicht **inline** erzeugen kann. Sie können diese Meldung ignorieren; die Funktion wird nicht-inline generiert.

Functions containing *reserviertes-Wort* are not expanded inline
[Funktion mit *reserviertes-Wort* kann nicht **inline** erweitert werden]
(Compiler, Warnung) Stehen die reservierten Worte **do**, **for**, **while**, **goto**, **switch**, **break**, **continue** und **case** in einer Funktion, kann diese auch dann nicht **inline** erweitert werden, wenn sie **inline** spezifiziert ist. Die Funktion ist zulässig, wird aber als **static** (nicht global) behandelt.

Functions may not be part of a struct or union
[Funktionen dürfen nicht Teil einer **struct** oder **union** sein]
(Compiler, Fehler) Hier wurde ein Feld eines **struct** oder **union** als Funktion anstatt als Zeiger auf eine Funktion deklariert. Nur in C++ sind Felder als Funktionen erlaubt.

General error
[Genereller Fehler]
(Linker, schwerer Fehler) Der Linker gibt über die Umstände zum Zeitpunkt des Auftretens eines unbekannten schweren Fehlers möglichst viele Informationen aus.

General error in library file *Dateiname* **in module** *Modul* **near module file offset 0x** *yyyyyyyy***.**
[Genereller Fehler in Bibliotheksdatei *Dateiname* in Modul *Modul*, in der Nähe des Moduldatei-Offsets 0x*yyyyyyyy*]
(Linker, schwerer Fehler) Der Linker gibt über die Umstände zum Zeitpunkt des Auftretens eines unbekannten schweren Fehlers möglichst viele Informationen aus.

General error in module *Modul* **near module file offset 0x** *yyyyyyyy***.**
[Genereller Fehler in Modul *Modul*, in der Nähe des Moduldatei-Offsets 0x*yyyyyyyy*]
(Linker, schwerer Fehler) Der Linker gibt über die Umstände zum Zeitpunkt des Auftretens eines unbekannten schweren Fehlers möglichst viele Informationen aus.

Global anonymous union not static
[Globale anonyme **union** ist nicht **static**]
(Compiler, Fehler) In C++ muß eine auf Dateiebene globale anonyme **union static** sein.

Goto bypasses initialization of a local variable
[**goto** umgeht die Initialisierung einer lokalen Variable]
(Compiler, Fehler) In C++ist es illegal, die Initialisierung einer lokalen Variable zu umgehen. In diesem Fall gibt es eine **goto**-Anweisung, die die Steuerung hinter die lokale Variable transferiert.

Goto statement missing label
[**goto**-Anweisung ohne Label]
(Compiler, Fehler) Nach dem Schlüsselwort **goto** muß ein Bezeichner stehen.

Group *Gruppe* **exceeds 64K**
[Gruppe *Gruppe* größer als 64 KByte]
(Linker, schwerer Fehler) Sie erhalten diese Meldung, wenn bei der Kombination der Gruppensegmente die Gruppe größer als 64 KByte wird.

Group overflowed maximum size: *Gruppe*
[Maximale Größe einer Gruppe überschritten: *Gruppe*]
(Compiler, Fehler) Die Gesamtgröße der Segmente einer Gruppe (z.B. DGROUP) überstieg 64 KByte.

Group *Gruppe1* **overlaps group** *Gruppe2*
[*Gruppe1* überschneidet *Gruppe2*]
(Linker, Warnung) Der Linker hat verschachtelte Gruppen gefunden. Die Warnung erhalten Sie nur bei Verwendung von Overlays oder beim Linken von Windows-Programmen.

***Spezifizierer* has already been included**
[*Spezifizierer* schon vorhanden]
(Compiler, Fehler) Der Spezifizierer taucht in der Deklaration öfter als einmal auf. Löschen oder ändern Sie einen dieser Spezifizierer.

Hexadecimal value contains more than 3 digits
[Hexadezimaler Wert hat mehr als drei Stellen]
(Compiler, Warnung) In älteren C-Versionen konnten hexadezimale Escape-Sequenzen nicht mehr als drei Stellen lang sein. Die neue ANSI-C-Norm erlaubt beliebig viele Stellen, solange der Wert noch in ein Byte paßt. Diese Warnung erscheint bei führenden Nullen in einer Escape-Sequenz ("\x00045"), weil ältere C-Versionen diesen String anders interpretieren würden.

Funktion1* hides virtual function *Funktion2
[*Funktion1* verbirgt die virtuelle *Funktion2*]
(Compiler, Warnung) Eine virtuelle Funktion einer Basisklasse wird normalerweise von der Deklaration einer abgeleiteten Klasse überschrieben. In diesem Fall macht eine Deklaration mit demselben Namen, aber mit verschiedenen Argumenttypen, die virtuellen Funktionen für weitere abgeleitete Klassen unzugänglich.

Identifier expected
[Bezeichner erwartet]
(Compiler, Fehler) Hier wird ein Bezeichner erwartet, tritt aber nicht auf. In C passiert das bei einer Parameterliste in einem Funktionskopf im alten Stil, nach den reservierten Worten **struct** und **union**, wenn keine Klammern folgen, oder als Feldname in einer Struktur oder Variante (außer bei Bitfeldern der Breite 0). In C++ wird ebenfalls ein Bezeichner erwartet, wenn eine Liste von Basisklassen auftritt, von der eine weitere Klasse abgeleitet wird, und zwar nach dem doppelten Doppelpunkt **::** und nach dem reservierten Begriff **operator**, wenn kein Operator folgt.

Identifier *Bezeichner* cannot have a type qualifier
[Bezeichner *Bezeichner* darf keinen Typ-Qualifizierer haben]
(Compiler, Fehler) Der C++ Qualifizierer *Klasse::Bezeichner* ist hier nicht erlaubt. Ein Qualifizierer ist nicht erlaubt bei **typedef**-Namen, Funktionsdeklarationen (außer Definitionen auf Dateiebene), vor lokalen Variablen oder Funktionsparametern oder bei Klassenelementen, außer als Qualifizierer der eigenen Klasse, was redundant, aber zulässig ist.

If statement missing (
[**if**-Anweisung: (direkt nach **if** fehlt]
(Compiler, Fehler) Öffnende Klammer (nach dem Schlüsselwort **if** fehlt.

If statement missing)
[**if**-Anweisung: abschließende) fehlt]
(Compiler, Fehler) Schließende Klammer) fehlt hinter dem Testausdruck einer *if*-Anweisung.

Illegal character ***Zeichen*** **(0x** ***Wert*****)**
[Unzulässiges Zeichen (0x*Wert*)]
(Compiler, Fehler) Der Compiler hat in der Eingabedatei ein unzulässiges Zeichen gefunden. Der hexadezimale Wert des betreffenden Zeichens wird ausgegeben. Dieser Fehler kann auch durch die Übergabe eines zusätzlichen Parameters an ein Funktionsmakro ausgelöst werden.

Illegal group definition: ***Gruppe*** **in module** ***Modul***
[Fehlerhafte Gruppendefinition: *Gruppe* im Modul *Modul*]
(Linker, schwerer Fehler) Dieser Fehler wird von einem fehlerhaften GRPDEF-Record in einer .OBJ-Datei ausgelöst. Dies kann an selbsterstellten .OBJ-Dateien oder an einem Fehler im für die Erstellung der .OBJ-Datei verwendeten Übersetzer liegen. Tritt der Fehler bei einer mit Turbo C++ erzeugten Datei auf, führen Sie die Compilierung noch einmal durch. Bleibt der Fehler bestehen, wenden Sie sich bitte an Borland GmbH.

Illegal initialization
[Unzulässige Initialisierung]
(Compiler, Fehler) Initialisierungen müssen in C entweder konstante Ausdrücke oder die Adresse einer globalen Variablen (**static** oder **extern**) plus/minus einer Konstanten sein.

Illegal octal digit
[Unzulässige Oktalziffer]
(Compiler, Fehler) Als oktale Konstanten sind nur die Zeichen *0* bis *7* als Ziffern erlaubt.

Illegal parameter to __emit__
[Unzulässiger Parameter für *__emit__*]
(Compiler, Fehler) Sie haben für *__emit__* ein Argument angegeben, das keine Konstante oder Adresse ist.

Illegal pointer subtraction
[Unzulässige Zeigersubtraktion]
(Compiler, Fehler) Die Subtraktion eines Zeigers von einem Nicht-Zeiger ist unzulässig.

Illegal structure operation
[Angegebene Operation mit Strukturen nicht zulässig]
(Compiler, Fehler) Strukturen können nur zusammen mit dem Punktoperator (.), dem Adreßoperator (**&**) und dem Zuweisungsoperator (**=**) verwendet oder als Argument von oder zu einer Funktion übergeben werden. In C oder C++ können Strukturen auch mit überladenen Operatoren verwendet werden. Der Compiler hat eine Struktur gefunden, die zusammen mit einem anderen Operator benutzt wird.

Illegal to take address of bit field
[Verwendung der Adresse eines Bitfeldes ist unzulässig]

(Compiler, Fehler) Es ist nicht gestattet, die Adresse eines Bitfeldes zu verwenden, bei anderen Feldern ist dies erlaubt.

Illegal use of floating point
[Verwendung von Gleitkommazahlen nicht erlaubt]
(Compiler, Fehler) In Schiebe-, Bit-, Konditional- (*?:*), Indirektions-(*) und weiteren Operatoren sind Gleitkommazahlen nicht als Operanden zugelassen. Der Compiler hat einen Gleitkommaoperanden bei einem nicht dafür zugelassenen Operator gefunden.

Illegal use of member pointer
[Verwendung des Elementzeigers nicht erlaubt]
(Compiler, Fehler) Zeiger auf Klassenelemente dürfen nur mit Zuweisungs-, Vergleichs-, .*-, ->*, *?:*-, *&&*- und ||-Operatoren verwendet oder als Argumente an Funktionen übergeben werden. Der Compiler hat einen Elementzeiger gefunden, der mit einem anderen Operator eingesetzt wurde.

Illegal use of pointer
[Zeiger an dieser Stelle nicht erlaubt]
(Compiler, Fehler) Zeiger können addiert, subtrahiert, zugeordnet, verglichen oder zusammen mit den Operatoren * und -> verwendet werden – alle anderen Operationen sind nicht zulässig.

Ill-formed pragma
[Unbekanntes Pragma]
(Compiler, Warnung) Dieses Pragma entspricht keinem, das der Borland C ++ Compiler kennt.

Implicit conversion of *Typ1* to *Typ2* not allowed
[Implizite Konvertierung von *Typ1* zu *Typ2* nicht zulässig]
(Compiler, Fehler) Wenn eine Elementfunktion einer Klasse mit einem Zeiger auf eine abgeleitete Klasse aufgerufen wird, muß der Wert des Zeigers implizit konvertiert werden, so daß auf die entsprechende Basisklasse gezeigt wird. In diesem Fall ist eine implizite Konvertierung nicht zulässig.

Imported reference from a VIRDEF to *Symbol*
[Importreferenz von einem VIRDEF auf *Symbol*]
(Linker, Fehler) Der Linker unterstützt augenblicklich keine Referenzen von VIRDEFs auf Symbole, die aus dynamischen Linkbibliotheken importiert werden. Wenn Sie eine Inline-Funktion in einer ausführbaren Datei haben, die ein statisches Datenelement einer Klasse in einer DLL referenziert, entfernen Sie die Funktion.

Improper use of typedef *Bezeichner*
[**typedef**-*Bezeichner* falsch angewendet]
(Compiler, Fehler) Der Quelltext verwendete ein **typedef**-Symbol, wo eine Variable in einem Ausdruck stehen sollte. Überprüfen Sie die Deklaration des Symbols und achten Sie auf mögliche Tippfehler.

***Dateiname* (*Zeilennummer*): Incompatible attribute**
[*Dateiname* (*Zeilennummer*): Unzulässiges Attribut]
(Linker, schwerer Fehler) Der Linker hat in einer CODE- oder DATA-Anweisung ein unzulässiges Attribut festgestellt. So können z.B. PRELOAD und LOADONCALL nicht gleichzeitig für dasselbe Segement gesetzt sein.

Incompatible type conversion
[Typumwandlung nicht möglich]
(Compiler, Fehler) Die angeforderte Typumwandlung ist nicht durchführbar.

Incorrect configuration file option: ***Option***
[Ungültiger Parameter in der Konfigurationsdatei: *Option*]
(Compiler, Fehler) Der Compiler erkennt den angegebenen Parameter nicht. Prüfen Sie auf fehlende Bindestriche (-).

Incorrect number format
[Falsches Zahlenformat]
(Compiler, Fehler) Der Compiler hat einen Dezimalpunkt in einer hexadezimalen Zahl gefunden.

Incorrect use of default
[Fehler in der **default**-Anweisung]
(Compiler, Fehler) Nach dem Schlüsselwort **default** wird kein Doppelpunkt (:) gefunden.

Initializing enumeration with ***Typ***
[Initialisierung der Aufzählung mit *Typ*]
(Compiler, Warnung) Eine Variable vom Typ **enum** wird mit einem falschen Typ initialisiert. Zum Beispiel:

```
enum count { zero, one, two } x = 2;
```

Dies führt zu einer Warnung, weil 2 ein Integerwert und nicht vom Typ **enum count** ist. Es zeugt von besserem Programmierstil, wenn ein **enum**-Wert anstelle eines Integerwertes zur Zuweisung oder Initialisierung von **enum**-Typen verwendet wird. Es handelt sich um einen Fehler, der aber als Warnung behandelt wird, damit bereits existierende Programme compiliert werden können.

Inline assembly not allowed in inline and template functions
[Inline-Assemblerausdrücke in Inline- oder Template-Funktionen nicht gestattet]
(Compiler, Fehler) Der Compiler kann diese Ausdrücke in C++ Inline- oder Template-Funktionen nicht bearbeiten. Sie könnten daraus ein Makro machen, die Speicherklasse **inline** entfernen oder den Inline-Assemblercode löschen.

Internal linker error ***Fehlercode***
[Interner Linker-Fehler *Fehlercode*]
(Linker, schwerer Fehler) Im internen Ablauf des Linkers ist ein Fehler aufgetreten. Normalerweise wird diese Meldung nicht auftreten. Wir haben sie hier der

Vollständigkeit halber für den Fall aufgeführt, daß Sie keine spezifischere Meldung erhalten. Bleibt dieser Fehler bestehen, notieren Sie sich bitte den Fehlercode, und wenden sich an die Borland GmbH.

Invalid combination of opcode and operands
[Unzulässige Kombination von Opcode und Operanden]
(Compiler, Fehler) Der integrierte Assembler akzeptiert diese Operandenkombination nicht. Folgende Gründe können dafür verantwortlich sein:

- Es sind zu viele oder zu wenige Operanden für diesen Assembler-Opcode vorhanden, z.B. *INC AX,BX* oder *MOV AX*.
- Die Anzahl der Operanden stimmt, aber ihre Typen oder ihre Reihenfolge entsprechen nicht dem Opcode, z.B. *DEC 1*, *MOV AX,CL* oder *MOV 1,AX*.

Invalid entry at *xxxxh*
[Unzulässiger Einsprung bei *xxxxh*]
(Linker, Warnung) Der Fehler tritt auf, wenn ein notwendiger Einsprung in der Einsprungtabelle einer ausführbaren Windows-Datei fehlt. Die Anwendung kann nicht im *Real Mode* ausgeführt werden, solange Programmtext und die Daten nicht korrigiert werden.

Invalid entry point offset
[Unzulässiger Einsprungpunkt-Offset]
(Linker, Fehler) Die Meldung erhalten Sie nur, wenn 32-Bit-Records gelinkt werden. Der Fehler tritt auf, wenn der Offset des ersten Einsprungpunktes des Programms die DOS-Beschränkung von 64 KByte überschreitet.

Invalid indirection
[Indirektion nicht zulässig]
(Compiler, Fehler) Der Indirektions-Operator (*) erfordert einen Nicht-**void**-Zeiger als Operand.

Invalid initial stack offset
[Unzulässiger Startwert für Stack-Offset]
(Linker, schwerer Fehler) Die Meldung erhalten Sie nur, wenn 32-Bit-Records gelinkt werden. Die Meldung besagt, daß der Startwert für den Stack-Zeiger die DOS-Beschränkung von 64 KByte überschreitet.

Invalid limit specified for code segment packing
(Unzulässige Größenangabe für das Zusammenfassen von Code-Segmenten)
(Linker, schwerer Fehler) Dieser Fehler tritt auf, wenn Sie die Option *Optionen/Linker/Einstellungen/Code-Segmente komprimieren* in der IDE verwendet und eine außerhalb des zulässigen Bereichs liegende Größenangabe für das Zusammenfassen gemacht haben. Die zulässige Größenangabe muß zwischen 1 und 65536 Bytes liegen. Die Voreinstellung ist 8192 Bytes.

Invalid macro argument separator
[Unzulässiges Trennzeichen zwischen Makroargumenten]
(Compiler, Fehler) Argumente in einer Makrodefinition müssen durch Kommas voneinander getrennt sein. Der Compiler hat ein anderes Zeichen nach einem Argumentnamen gefunden.

Invalid page size value ignored
[Unzulässiger Wert für die Seitengröße wurde ignoriert]
(Bibliotheksverwaltung, Warnung) Die angegebene Seitengröße ist nicht zulässig. Die Seitengröße muß eine Potenz von 2 sein. Sie darf nicht kleiner als 16 und nicht größer als 32768 sein.

Invalid pointer addition
[Addition zweier Zeiger nicht erlaubt]
(Compiler, Fehler) Es wurde versucht, zwei Zeiger zu addieren.

Invalid register combination (e.g. [BP+BX])
[Unzulässige Registerkombination (z.B. [BP+BX])]
(Compiler, Fehler) Der integrierte Assembler hat eine unzulässige Kombination von Registern aufgefunden. Zulässige Index-Registerkombinationen sind [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI] und [BP+DI]. Andere Kombinationen, wie [AX], [BP+BX] und [SI+BX], sind nicht erlaubt.
Hinweis: Lokale Variablen (in Prozeduren und Funktionen deklarierte Variablen) befinden sich normalerweise auf dem Stack. Der Zugriff erfolgt über das BP-Register. Der Assembler fügt automatisch [BP] in die Referenzen auf derartige Variablen ein. Auch wenn ein Konstrukt wie *Local*[*BX*] gültig aussieht (wobei *local* eine lokale Variable ist), ist dies nicht der Fall. Der endgültige Operand würde zu *Local*[*BP+BX*].

Invalid segment definition in module ***Modul***
[Unzulässige Segmentdefinition im Modul *Modul*]
(Linker, schwerer Fehler) Diese Meldung erhalten Sie grundsätzlich nur dann, wenn ein Compiler eine »unsinnige« .OBJ-Datei erstellt hat. Wenn dies bei einer mit Turbo C++erzeugten Datei der Fall ist, so führen Sie die Compilierung erneut durch. Bleibt das Problem bestehen, dann wenden Sie sich bitte an die Borland GmbH.

Invalid size specified for segment alignment
[Unzulässige Größe für Segmentausrichtung]
(Linker, Fehler) Für die Option *Optionen/Linker/Einstellungen/Segment* in der IDE wurde ein unzulässiger Wert festgelegt. Die angegebene Größe muß ein ganzzahliges Vielfaches von 2 und kleiner als 64 KByte sein. Gebräuchliche Werte sind 16 und 512. Der Fehler tritt nur beim Linken von Windows-Programmen auf.

Invalid template argument list
[Ungültige Parameterliste in einer Template-Deklaration]
(Compiler, Fehler) In einer Template-Deklaration muß auf das Schlüsselwort

template eine Liste mit formalen Parametern folgen, die in spitzen Klammern (<und >) steht. Die Meldung wird ausgegeben, wenn eine ungültige Parameterliste vorhanden ist.

Invalid template qualified name ***Template::Name***
[Unzulässiger qualifizierter Template-Name *Schablone::Name*]
(Compiler, Fehler) Bei der Definition eines Template-Klassenelements müssen die Argumente im Template-Klassennamen, der als linker Operand im *::*-Operator fungiert, mit den formalen Argumenten der Template-Klasse übereinstimmen. Beispiel:

```
template <class T> class X
{
   void f();
};
template <class T> void X<T>::f() {}
```

Folgendes ist nicht zulässig:

```
template <class T> void X<int>::f() {}
```

Invalid use of dot
[Unzulässige Verwendung des Operators .]
(Compiler, Fehler) Auf den Operator Punkt (.) muß unmittelbar der Bezeichner folgen.

Invalid use of template ***Template***
[Unzulässige Verwendung der Template *Schablone*]
(Compiler, Fehler) Außerhalb einer Template-Definition ist die Verwendung eines Template-Klassennamens ohne Argumente nicht zulässig. So können Sie beispielsweise *vector<int>*, aber nicht *vector* angeben.

Irreducible expression tree
[Interner Compilerfehler bei der Auswertung eines Ausdrucks]
(Compiler, schwerer Fehler) Hier handelt es sich um einen Compilerfehler. Ein Ausdruck in der angegeben Zeile verhindert, daß der Code-Generator Code erzeugen kann. Lassen Sie den Ausdruck weg. Sollten Sie jemals diesen Fehler erhalten, kontaktieren Sie bitte Borland GmbH.

Basis **is an indirect virtual base class of** ***Klasse***
[*Basis* ist eine indirekte virtuelle Basisklasse von *Klasse*]
(Compiler, Fehler) Ein Zeiger auf ein C++ Element einer virtuellen Basisklasse kann nicht angelegt werden. Es wurde versucht, einen derartigen Zeiger zu erstellen (entweder direkt oder über eine Typumwandlung).

Bezeichner **is assigned a value which is never used**
[*Bezeichner* wird ein Wert zugewiesen, der nie verwendet wird]
(Compiler, Warnung) Die Variable erscheint in einer Zuweisung, wird aber nirgends in der gerade beendeten Funktion verwendet. Die Warnung wird erst ausgegeben, wenn der Compiler eine rechte geschweifte Klammer (}) findet.

***Bezeichner* is declared both as external and static**
[*Bezeichner* ist sowohl **extern** als auch **static** deklariert]
(Compiler, Warnung) Der Bezeichner trat in einer Deklaration auf, die ihn implizit oder explizit als global oder **extern** kennzeichnete, aber auch in einer **static** Deklaration. Er gilt jetzt als **static**. Überprüfen Sie alle Deklarationen für diesen Bezeichner.

Symbol* is duplicated in module *Modul
[Symbol im Modul *Modul* mehrfach enthalten]
(Linker, Fehler oder Warnung) Sie erhalten diese Meldung, wenn es zwischen zwei Symbolen (öffentlichen oder gemeinsamen) in einem Modul zu Problemen kommt. Ein Fehler tritt auf, wenn beide Symbole in einer .OBJ-Dateien stehen. Zu einer Warnung kommt es dann, wenn der Linker die beiden identischen Symbole in einer Bibliothek vorfindet. In diesem Fall verwendet der Linker die erste Definition.

Konstruktor* is not a base class of *Klasse
[*Konstruktor* ist keine Basisklasse von *Klasse*]
(Compiler, Fehler) Ein C++ Klassenkonstruktor *Klasse* versucht einen Basisklassenkonstruktor *Konstruktor* aufzurufen, oder Sie versuchen die Zugriffsrechte von *Klasse::Konstruktor* zu verändern. *Konstruktor* ist keine Basisklasse von *Klasse*. Bitte überprüfen Sie die Deklarationen.

Bezeichner* is not a member of *Struktur
[*Bezeichner* ist kein Element von *Struktur*]
(Compiler, Fehler) Sie versuchen fälschlicherweise, einen Bezeichner als Element von *Struktur* zu behandeln. Prüfen Sie die Deklarationen nach.

***Bezeichner* is not a non-static data member and can't be initialized here**
[*Bezeichner* ist kein nicht-statisches Datenelement und kann hier nicht initialisiert werden]
(Compiler, Fehler) Nur Datenelemente können mit dem Initialisierer in einem Konstruktor initialisiert werden, hier lag aber ein statisches oder ein Funktionselement vor.

***Bezeichner* is not a parameter**
[*Bezeichner* ist kein Parameter]
(Compiler, Fehler) Im Abschnitt Parameterdeklaration einer Funktionsdefinition alten Stils wird der *Bezeichner* deklariert, aber nicht als Parameter aufgelistet. Entfernen Sie die Deklaration oder ergänzen Sie *Bezeichner* als Parameter.

Bezeichner* is not a public base class of *Klassentyp
[*Bezeichner* ist keine öffentliche Basisklasse von *Klassentyp*]
(Compiler, Fehler) Der rechte Operand von .*, ->* oder *::operator* ist kein Zeiger auf ein Element einer Klasse, die entweder eine eindeutig zugreifbare Basisklasse des linken Operanden des Klassentyps oder mit einer derartigen Klasse identisch ist.

***Element* is not accessible**
[*Element* nicht zugänglich]
(Compiler, Fehler) Sie versuchen, auf ein Element einer C++Klasse zuzugreifen, das **private** oder **protected** und von dieser Funktion nicht referenzierbar ist. Dies geschieht manchmal, wenn eine zugängliche überladene Elementfunktion (oder ein Konstruktor) aufgerufen werden soll, die Argumente jedoch einer unzugänglichen Funktion entsprechen. Mögliches Überladen wird immer vor den Zugriffsrechten geprüft. Wandeln Sie einen oder mehrere Parameter um, um die gewünschte zugreifbare Funktion auszuwählen.

Last parameter of *Operator* must have type int
[Der letzte Parameter von *Operator* muß vom Typ **int** sein]
(Compiler, Fehler) Wenn ein Suffix-Operator *operator*++ oder *operator*-- deklariert wird, muß der letzte Parameter vom Typ **int** sein.

library contains COMFEF records extended dictionary not created
[Die Bibliothek enthält COMDEF-Records das erweiterte Verzeichnis wurde nicht angelegt.]
(Bibliotheksverwaltung, Warnung) Ein Objekt-Record, der einen COMDEF-Record enthält, wurde zu einer Bibliothek hinzugefügt. Dies aber ist nicht mit der Erstellung eines erweiterten Verzeichnisses verträglich.

library too large, restart with library page size *Größe*
[Bibliothek zu groß, starten Sie erneut mit der Bibliotheksseitengröße *Größe*]
(Bibliotheksverwaltung, Fehler) Die zu erzeugende Bibliothek kann nicht mit der aktuellen Seitengröße aufgebaut werden. Sie können die Seitengröße im Dialogfenster *Bibliotheksverwaltung* des Menüs *Optionen/Bibliotheksverwaltung* festlegen.

Limit of 254 segments for new executable file exceeded
[Die Obergrenze von 254 Segmenten für die neue ausführbare Datei wurde überschritten]
(Linker, schwerer Fehler) Das neue Dateiformat für ausführbare Dateien läßt maximal 254 Segmente zu. Überprüfen Sie die .MAP-Datei. Normalerweise gibt es zwei Ursachen für diesen Fehler: Wenn die Anwendung das Speichermodell *Large* verwendet, könnte die Zusammenfassungsgrenze für Code-Segmente zu klein sein, so daß es zu viele Code-Segemente gibt. Die Erhöhung der Zusammenfassungsgrenze löst das Problem. Die zweite Fehlermöglichkeit ist, daß Sie viele **far**-Datensegmente haben, die nur wenige Bytes enthalten. Die .MAP-Datei teilt Ihnen dies mit. Verringern Sie die Anzahl der **far**-Datensegmente.

Linkage specification not allowed
[Bindungsspezifikation nicht zulässig]
(Compiler, Fehler) Bindungsspezifikationen wie **extern** "C" sind nur auf Dateiebene erlaubt. Plazieren Sie diese Funktionsdeklaration auf Dateiebene.

Linker stack overflow
[Linker-Stack-Überlauf]
(Linker, schwerer Fehler) Der Linker setzt eine rekursive Prozedur für die Markierung der Module ein, die in ein ausführbares Bild aus Bibliotheken eingebunden werden sollen. Diese Prozedur kann im Extremfall Stack-Überläufe produzieren. Wenn Sie diese Fehlermeldung erhalten, sollten Sie einige Module aus Bibliotheken entfernen, diese mit den Objektdateien in den Linkvorgang einbinden und anschließend den Linkvorgang neu starten.

Lvalue required
[L-Wert erforderlich]
(Compiler, Fehler) Die linke Seite eines Zuweisungsoperators muß ein adreßierbarer Ausdruck sein. Dies schließt numerische und Zeigervariablen, Referenzen auf Strukturfelder, Indirektionen durch Zeiger oder ein indiziertes Array-Element ein.

Macro argument syntax error
[Makro-Argument: Syntaxfehler]
(Compiler, Fehler) Ein Argument in einer Makrodefinition muß ein Bezeichner sein. Der Compiler hat ein für Bezeichner ungültiges Zeichen angetroffen, obwohl an diesem Punkt ein Argument erwartet wurde.

Macro expansion too long
[Makro-Erweiterung zu lang]
(Compiler, Fehler) Ein Makro kann nicht zu mehr als 4096 Zeichen erweitert werden.

Matching base class function for *Funktion* has different dispatch number
[Die entsprechende Basisklassenfunktion für *Funktion* hat eine andere Zuordnungsnummer]
(Compiler, Fehler) Wenn eine DDVT-Funktion in einer abgeleiteten Klasse deklariert ist, muß die entsprechende Funktion der Basisklasse dieselbe Zuordnungsnummer wie die abgeleitete Funktion haben.

Matching base class function for *Funktion* is not dynamic
[Die entsprechende Basisklassenfunktion für *Funktion* ist nicht dynamisch]
(Compiler, Fehler) Wenn eine DDVT-Funktion in einer abgeleiteten Klasse deklariert ist, muß die entsprechende Funktion der Basisklasse dynamisch sein.

Member function must be called or its adress taken
[Die Elementfunktion muß aufgerufen oder ihre Adresse ermittelt werden]
(Compiler, Fehler) Wenn in einem Ausdruck eine Elementfunktion verwendet wird, muß diese aufgerufen oder ihre Adresse mit dem Operator **&** ermittelt werden. In diesem Fall wurde eine Elementfunktion in einem unzulässigen Kontext eingesetzt.

Member identifier expected
[Es wird ein Element-Bezeichner erwartet]
(Compiler, Fehler) Der Name einer Struktur oder eines C++ Klassenelements wurde erwartet, aber nicht gefunden. Die rechte Seite eines Punkt- oder Pfeil-Operators (. oder ->) muß der Name eines Elements in der Struktur oder der Name einer Klasse auf der linken Seite des Operators sein.

Member is ambigous: ***Element1*** **and** ***Element2***
[Element ist mehrdeutig: *Element1* und *Element2*]
(Compiler, Fehler) Sie müssen die Elementreferenz mit dem entsprechenden Basisklassennamen referenzieren. In der C++ Klasse *Klasse* kann das Element *Element* in mehreren Basisklassen ohne genaue Zuordnung vorhanden sein. Dies ist nur bei Mehrfachvererbungen der Fall, in denen der Elementname in jeder Basisklasse nicht durch denselben Elementnamen in einer abgeleiteten Klasse mit demselben Pfad verborgen wird.

Die Regeln von C++ erfordern, daß dieser Test auf Mehrdeutigkeit vor der Überprüfung der Zugriffsrechte (**private**, **protected**, **public**) durchgeführt wird. Aus diesem Grund können Sie diese Meldung auch erhalten, wenn nur auf ein (oder kein) Element zugegriffen werden kann.

Member ***Element*** **cannot be used without an object**
[Element *Element* kann nicht ohne Objekt verwendet werden]
(Compiler, Fehler) Diese Meldung besagt, daß im Quelltext *Klasse::Element* enthalten ist, wobei *Element* ein normales (nicht-statisches) Element ist, und es gibt keine Klasse, die mit diesem Element verknüpft werden kann. *obj.Klasse::Element* ist zulässig, *Klasse::Element* jedoch nicht.

Member ***Element*** **has the same name as its class**
[Element *Element* hat denselben Namen wie seine Klasse]
(Compiler, Fehler) Ein statisches Datenelement, ein Aufzählungstyp, ein Element einer anonymen Variante oder ein geschachtelter Typ dürfen nicht denselben Namen wie ihre Klasse haben. Nur Elementfunktionen oder nicht-statische Elemente können denselben Namen wie ihre Klassen tragen.

Member ***Element*** **is initialized more than once**
[Element *Element* mehrmals initialisiert]
(Compiler, Fehler) In einem C++Klassenkonstruktor enthält die Initialisierungsliste nach dem Konstruktorkopf denselben Elementnamen mehrmals.

Member pointer required on right side of .* oder ->*
[Auf der rechten Seite der Operatoren .* oder ->* muß ein Elementzeiger stehen]
(Compiler, Fehler) Die rechte Seite der Operatoren .* oder ->* muß als Zeiger auf ein Element der Klasse deklariert sein, die auf der linken Seite des Operators angegeben ist. Wenn diese Fehlermeldung ausgegeben wird, ist die rechte Seite kein Elementzeiger.

Memory full listing truncated!
[Listendatei nicht vollständig]
(Bibliotheksverwaltung, Warnung) Der Bibliotheksverwaltung steht bei der Erstellung einer Listendatei für eine Bibliothek kein Speicherplatz mehr zur Verfügung. Es wird eine Listendatei angelegt, diese ist jedoch nicht vollständig.

Memory reference expected
[Speicher-Referenz erwartet]
(Compiler, Fehler) Der integrierte Assembler benötigt eine Speicherreferenz. Ein häufiger Grund liegt darin, daß die eckigen Klammern um den Operanden eines Index-Registers fehlen, z.B. MOV AX,BX+SI anstelle von MOV AX,[BX+SI].

Misplaced break
[Falsch positionierte **break**-Anweisung]
(Compiler, Fehler) Der Compiler hat eine **break**-Anweisung außerhalb einer **switch**-Anweisung oder Schleife gefunden.

Misplaced continue
[Falsch positionierte **continue**-Anweisung]
(Compiler, Fehler) Der Compiler hat eine **continue**-Anweisung außerhalb einer **switch**-Anweisung oder Schleife gefunden.

Misplaced decimal point
[Falsch positionierter Dezimalpunkt]
(Compiler, Fehler) Der Compiler hat im Exponenten von Gleitkommazahlen einen Dezimalpunkt gefunden.

Misplaced elif directive
[Falsch positionierte #*elif*-Anweisung]
(Compiler, Fehler) Der Compiler hat eine #*elif*-Anweisung ohne vorangehende #*if*-, #*ifdef*- oder #*ifndef*-Anweisung gefunden.

Misplaced else
[Falsch positionierte **else**-Anweisung]
(Compiler, Fehler) Der Compiler hat eine **else-**Anweisung ohne dazugehöriges **if** gefunden. Ein überzähliges **else** kann zu dieser Meldung geführt haben, aber auch ein überzähliger Strichpunkt, überzählige geschweifte Klammern oder ein Syntaxfehler in einer vorausgehenden **if**-Anweisung.

Misplaced else directive
[Falsch positionierte #*else*-Anweisung]
(Compiler, Fehler) Der Compiler hat eine #*else*-Anweisung ohne vorangehendes #*if*, #*ifdef* oder #*ifndef* gefunden.

Misplaced endif directive
[Falsch positionierte #*endif*-Anweisung]
(Compiler, Fehler) Der Compiler hat eine #*endif*-Anweisung ohne vorangehendes #*if*, #*ifdef* oder #*ifndef* gefunden.

***Dateiname* (*Zeilennummer*): Missing internal name**
[*Dateiname* (*Zeilennummer*): Interner Name fehlt]
(Linker, schwerer Fehler) Im IMPORTS-Bereich der Moduldefinitionsdatei ist eine Referenz zu einem Einsprung über einen Modulnamen und einen Ordinalwert angegeben. Wird ein Einsprung durch einen Ordinalwert spezifiziert, muß dieser Importdefinition ein interner Name zugewiesen sein. Diesen internen Namen verwendet das Programm, um auf die importierte Definition Bezug zu nehmen. Die Syntax in der Moduldefinitionsdatei sollte folgendermaßen aussehen:

<InternerName>=<Modulname>.<Ordinal>

Mixing pointers to signed and unsigned char
[Vermischung von Zeigern auf **signed** und **unsigned char**]
(Compiler, Warnung) Diese Warnung erhalten Sie bei der Konvertierung eines Zeigers auf **signed char** in **unsigned char** (oder umgekehrt), wenn keine explizite Typumwandlung angegeben ist. Derartige Umwandlungen sind zwar streng genommen unzulässig, funktionieren aber bei 8086-Prozessoren normalerweise ohne Probleme.

Multiple base classes require explicit class names
[Mehrfache Basisklassen erfordern explizite Klassennamen]
(Compiler, Fehler) In einem C++ Klassenkonstruktor muß jeder Aufruf des Konstruktors der Basisklasse im Konstruktorkopf den Namen der Basisklasse enthalten, wenn mehr als eine unmittelbare Basisklasse existiert.

Multiple declaration for *Bezeichner*
[Bezeichner mehrfach deklariert]
(Compiler, Fehler) Der Bezeichner wurde fälschlicherweise mehrfach deklariert. Dies kann aus widersprüchlichen Angaben wie *int a; double a;* herrühren, oder aus einer zweimal verschieden deklarierten Funktion, einem wiederholten Sprungziel (Label) in derselben Funktion oder einer wiederholten Deklaration außer einer **extern**-Funktion oder einer einfachen Variablen (in C).

***Bezeichner* must be a member function**
[*Bezeichner* muß Elementfunktion sein]
(Compiler, Fehler) Die meisten Operatorfunktionen in C++ können Elemente von Klassen oder gewöhnliche Nicht-Elementfunktionen sein. Manche *müssen* Elemente von Klassen sein, wie *operator =*, *operator ->*, *operator*() und Typumwandlungen. Diese Operatorfunktion ist keine Elementfunktion, sollte es aber sein.

***Bezeichner* must be a member function or have a parameter of class type**
[*Bezeichner* muß Elementfunktion sein oder einen Parameter des Typs **class** enthalten]
(Compiler, Fehler) Die meisten Operatorfunktionen in C++ müssen einen expliziten oder impliziten Parameter des Typs **class** aufweisen. Diese Operator-

funktion wurde außerhalb einer Klasse deklariert und enthält keinen expliziten Parameter vom Typ **class**.

***Bezeichner* must be previously defined class or struct**
[*Bezeichner* muß vorher als **class** oder **struct** definiert werden]
(Compiler, Fehler) Sie versuchen, einen Bezeichner als Basisklasse zu definieren. Entweder ist er keine Klasse oder nicht vollständig definiert. Ändern Sie den Namen oder die Deklarationen.

***Bezeichner* must be a previously defined enumeration tag**
[*Bezeichner* muß vorher als **enum**-»Tag« definiert werden]
(Compiler, Fehler) Sie versuchen einen Bezeichner als »Tag« eines **enum**-Typs anzusprechen, aber er wurde so nicht deklariert. Ändern Sie den Namen oder die Deklarationen.

***Funktion* must be declared with no parameters**
[*Funktion* muß ohne Parameter deklariert werden]
(Compiler, Fehler) Diese C++ Operatorfunktion ist fälschlicherweise mit Parametern deklariert.

***Funktion* must be declared with one parameter**
[*Funktion* muß mit einem Parameter deklariert werden]
(Compiler, Fehler) Diese C++ Operatorfunktion ist fälschlicherweise mit mehr als einem Parameter deklariert.

***Operator* must be declared with one or no parameters**
[*Operator* muß mit einem oder ohne Parameter deklariert werden]
(Compiler, Fehler) Wenn *operator*++ oder *operator*-- als Elementfunktion deklariert ist, darf sie keinen Parameter (für die Präfix-Version des Operators) oder muß einen Parameter vom Typ **int** haben (für die Suffix-Version).

***Operator* must be declared with one or two parameters**
[*Operator* muß mit einem oder zwei Parametern deklariert werden]
(Compiler, Fehler) Wenn *operator*++ oder *operator*-- als Nicht-Elementfunktion deklariert ist, muß sie einen Parameter (für die Präfix-Version des Operators) oder zwei Parameter haben (für die Suffix-Version).

***Funktion* must be declared with two parameters**
[*Funktion* muß mit zwei Parametern deklariert werden]
(Compiler, Fehler) Diese C++ Operatorfunktion ist fälschlicherweise nicht mit zwei Parametern deklariert.

Must take address of memory location
[Keine Adresse für Speicherposition vorhanden]
(Compiler, Fehler) In der Quelltextdatei ist ein Adreßoperator (*&*) mit einem unzulässigen Ausdruck vorhanden, z.B. einer Registervariablen.

Need an identifier to declare
[Bezeichner für Deklaration erforderlich]
(Compiler, Fehler) In diesem Zusammenhang wird ein Bezeichner erwartet, um die Deklaration zu vervollständigen. Es kann z.B. ein typedef ohne Name oder ein überzähliger Strichpunkt auf Dateiebene vorliegen. In C++ kann es auch ein Klassenname sein, der fälschlich für eine anderere Bezeichnerart verwendet wird.

No : following the ?
[Kein Doppelpunkt nach Fragezeichen]
(Compiler, Fehler) In diesem Ausdruck sind die Operatoren ? und : falsch plaziert. Vielleicht fehlt der Doppelpunkt, oder Klammern sind falsch verschachtelt oder fehlen.

No automatic data segment
[Automatisches Datensegment nicht vorhanden]
(Link, Warnung) Die Gruppe DGROUP ist nicht vorhanden. DGROUP wird in den Initialisierungsdateien von Turbo C++ definiert. Wenn Sie diese Meldung erhalten, wurde keine Initialisierungsdatei gelinkt (und DGROUP wurde auch von Ihrem Programm nicht definiert). Windows benutzt DGROUP zum Auffinden des lokalen Datensegments. DGROUP muß für Windows-Anwendugen (jedoch nicht für dynamische Linkbibliotheken) vorhanden sein, außer in der Moduldefinitionsdatei ist die Option DATA NONE gesetzt.

No base class to initialize
[Basisklasse fehlt]
(Compiler, Fehler) Dieser C++ Klassenkonstruktor versucht implizit den Konstruktor einer Basisklasse aufzurufen. Für diese Klasse wurde aber keine Basisklasse deklariert. Prüfen Sie die Deklarationen.

No declaration for function ***Funktion***
[Keine Deklaration für Funktion *Funktion*]
(Compiler, Warnung) Diese Warnung erhalten Sie, wenn Sie eine Funktion aufrufen, die vorher nicht deklariert wurde. In C kann eine Funktion ohne Prototyp (wie `int func();`) deklariert werden. In C++ ist jede Funktionsdeklaration auch ein Prototyp; das Beispiel ist äquivalent zu `int func(void);`. Die Deklaration kann hier im klassischen oder im modernen Stil erfolgen.

No file name ending
[Dateiname nicht korrekt abgeschlossen]
(Compiler, Fehler) Hinter dem Dateinamen in einer #*include*-Anweisung fehlt die erforderliche schließende spitze Klammer oder das abschließende Anführungszeichen.

No module definition file specified: using defaults
[Keine Moduldefinitionsdatei angegeben; Verwendung der Voreinstellungen]
(Linker, Warnung) Der Linker wurde mit einer der Windows-Optionen aufgerufen, es wurde aber keine Moduldefinitionsdatei angegeben.

No programm starting address defined
[Keine Startadresse für das Programm definiert]
(Linker, Warnung) Es ist kein Modul vorhanden, in dem die Startadresse für das Programm definiert ist. Der Fehler tritt insbesondere dann auf, wenn vergessen wurde, das Initialisierungsmodul C0*x*.OBJ zu linken. Beim Linken einer dynamischen Linkbibliothek für Windows sollte diese Meldung allerdings nicht ausgegeben werden.

Non-const function ***Funktion*** **called for const object**
[Nicht-konstante Funktion *Funktion* für konstantes Objekt aufgerufen]
(Compiler, Warnung) Eine nicht-konstante Elementfunktion wurde für ein Objekt vom Typ **const** aufgerufen. Dies ist ein Fehler, der aber auf eine Warnung herabgestuft wurde, damit schon existierende Programme compiliert werden können.

Nonportable pointer comparison
[Nicht portierbarer Zeigervergleich]
(Compiler, Warnung) Ihr Quelltext enthält einen Vergleich von einem Zeiger mit einem Nicht-Zeiger (der nicht 0 ist). Verwenden Sie eine Typumwandlung, wenn Sie diese Warnung verhindern wollen und der Vergleich korrekt ist.

Nonportable pointer conversion
[Typfehler bei Zeigerumwandlung]
(Compiler, Fehler) Es ist hier eine implizite Umwandlung von einem Zeiger in einen Integertyp verlangt, die Typen haben aber nicht dieselbe Größe. Nur eine explizite Typumwandlung ist möglich. Die Umwandlung ist eventuell unsinnig, überprüfen Sie den Vorgang noch einmal.

Nonportable pointer conversion
[Nicht portierbare Zeigerumwandlung]
(Compiler, Warnung) Ein Integerwert ungleich Null wird verwendet, wo ein Zeiger benötigt wird oder umgekehrt. Die Größe von Integerwert und Zeiger sind identisch. Wenn Sie dies wirklich wollen, verwenden Sie eine explizite Typumwandlung.

Nontype template argument must be of scalar type
[Ein nicht typisiertes Template-Argument muß ein skalarer Typ sein]
(Compiler, Fehler) Ein formales nicht typisiertes Template-Argument muß ein skalarer Typ sein; es kann vom Typ **int**, **enum** oder ein Zeigertyp sein.

Non-virtual function ***Funktion*** **declared pure**
[Nicht-virtuelle Funktion *Funktion* als abstrakt deklariert]
(Compiler, Fehler) Nur virtuelle Funktionen können als abstrakte Funktionen deklariert werden, weil abgeleitete Klassen sie überschreiben können müssen.

Non-volatile function ***Funktion*** **called for volatile object**
[Eine nicht-**volatile** Funktion wurde für ein Objekt vom Typ **volatile** aufgerufen]

(Compiler, Warnung) In C++ wurde eine Klassenelementfunktion für ein **volatile**-Klassenobjekt aufgerufen, die Funktion ist aber nicht mit **volatile** nach dem Funktionskopf deklariert.

Not an allowed type
[Typ ist nicht zulässig]
(Compiler, Fehler) Sie haben einen unzulässigen Typ deklariert, beispielsweise eine Funktion, die eine Funktion oder ein Array zurückliefert.

Not enough memory
[Nicht genügend Speicherplatz]
(Linker, schwerer Fehler) Es ist nicht genügend Speicher für den Betrieb des Linkers vorhanden. Schließen Sie eine oder mehrere Anwendungen und starten Sie den Linkvorgang erneut.

Not enough memory for command-line buffer
[Nicht genügend Speicheplatz für Kommandozeilenpuffer]
(Bibliotheksverwaltung, Fehler) Diese Meldung erhalten Sie, wenn für die Bibliotheksverwaltung nicht mehr genügend Speicherplatz vorhanden ist.

Numeric constant too large
[Numerische Konstante zu groß]
(Compiler, Fehler) Es können keine String- oder Zeichen-Escape-Sequenzen erzeugt werden, die größer als hex *\xFF* oder oktal *\377* sind. Zwei-Byte-Zeichenkonstanten können mit Hilfe eines zweiten Backslash angegeben werden. Zum Beispiel steht *\x0D\x0A* für eine Zwei-Byte-Konstante. Ein numerisches Literal direkt nach einer Escape-Sequenz sollte so abgetrennt werden:

```
printf("\x0D" "12345");
```

Damit wird ein CR ausgegeben und anschließend *12345*.

Object module *Dateiname* **is invalid**
[Objektmodul *Dateiname* ist ungültig]
(Bibliotheksverwaltung, Fehler) Die Bibliotheksverwaltung konnte den Kopf-Record des in die Bibliothek eingefügten Objektmoduls nicht interpretieren. Sie nimmt deshalb an, daß es sich um ein ungültiges Modul handelt.

Objects of type *Typ* **cannot be initialized with { }**
[Objekte vom Typ *Typ* können nicht mit { } initialisiert werden]
(Compiler, Fehler) Normale C-Strukturen werden mit Werten initialisiert, die innerhalb von geschweiften Klammern stehen. C++ Klassen können nur mit Konstruktoren initialisiert werden, wenn die Klasse über Konstruktoren, private Elemente, virtuelle Funktionen oder virtuelle Basisklassen verfügt.

Only member functions may be 'const'or 'volatile'
[Nur Elementfunktionen können vom Typ **const** oder **volatile** sein]
(Compiler, Fehler) Etwas anderes als eine Elementfunktion wurde als **const** und/oder **volatile** deklariert.

Only one of a set of overloaded functions can be *"C"*
[Nur eine der überladenen Funktionen darf als extern "C" deklariert werden]
(Compiler, Fehler) Alle C++Funktionen werden per Voreinstellung überladen, wobei der Compiler jeder Funktion einen neuen Namen zuweist. Wollen Sie, daß die Funktion ihren alten Namen behält (durch Deklaration als extern "C"), dürfen Sie das nur bei einer Funktion aus einer Gruppe mit gleichem Namen tun. Sonst würde der Linker mehr als eine globale Funktion mit demselben Namen finden.

Operand of delete must be non-const pointer
[Der Operand von **delete** muß ein nicht-konstanter Zeiger sein]
(Compiler, Fehler) Ein konstanter Zeigerwert darf nicht mit dem Operator **delete** gelöscht werden.

Operator [] missing]
[Im Operator [] fehlt]]
(Compiler, Fehler) Der C++Operator **[]** wurde als **operator [** deklariert. Ergänzen Sie die fehlende eckige Klammer oder ändern Sie die Deklaration.

Operator -> must return a pointer or a class
[**operator->** muß einen Zeiger oder eine Klasse liefern]
(Compiler, Fehler) Die C++ Funktion **operator->** muß als Ergebnis entweder eine Klasse oder einen Zeiger auf **class** (oder **struct** oder **union**) liefern. Es muß sich in jedem Fall um etwas handeln, auf das sich der Operator **->** beziehen kann.

operator delete must return void
[Der **delete**-Operator muß **void** zurückliefern]
(Compiler, Fehler) Dieser überladene C++ Operator **delete** ist hier anders deklariert.

Operator must be declared as function
[Operator muß als Funktion deklariert sein]
(Compiler, Fehler) Ein überladener Operator wurde nicht als Funktion deklariert.

operator new must have an initial parameter of type size_t
[Der Operator **new** muß einen Parameter des Typs *size_t* haben]
(Compiler, Fehler) Der Operator **new** kann mit einer beliebigen Anzahl von Parametern deklariert werden, muß aber mindestens einen Parameter aufweisen, der die Größe des zuzuweisenden Speichers angibt.

operator new must return an object of type void*
[Der **new**-Oeprator muß ein Objekt vom Typ **void** zurückliefern]
(Compiler, Fehler) Dieser überladene C++Operator **new** war hier anders deklariert.

Operators may not have default argument values
[Operatoren dürfen keine voreingestellten Argumentwerte haben]
(Compiler, Fehler) Überladene Operatoren dürfen keine voreingestellten Argumentwerte haben.

Out of memory
[Nicht genügend Arbeitsspeicher]
(Compiler, schwerer Fehler) Der gesamte Arbeitsspeicher wurde verbraucht. Die Datei sollte auf einem Rechner mit größerem Arbeitsspeicher compiliert werden. Unter Windows sollten Sie eine oder mehrere Anwendungen schließen, um Speicherplatz freizugeben.

Out of memory
[Zu wenig Speicher]
(Bibliotheksverwaltung, Fehler) Es gibt verschiedene Gründe, aus denen die Bibliotheksverwaltung oder Turbo C++ beim Aufbau einer Bibliothek zu wenig Speicher zur Verfügung haben. In vielen Fällen wird eine detailliertere Fehlermeldung ausgegeben. »Out of memory« fungiert als allgemeine Meldung in Situationen, in denen zu wenig Speicherplatz verfügbar ist. Unter Windows können Sie eine oder mehrere Anwendungen schließen, um Speicherplatz freizugeben.
Wenn Sie diese Meldung bei zu groß gewordenen öffentlichen Symboltabellen erhalten, müssen Sie Speicherplatz freigeben. Für die IDE schließen Sie Editoren. Unter Windows können Sie eine oder mehrere Anwendungen schließen, um Speicherplatz freizugeben.

Out of memory
[Zu wenig Speicher]
(Linker, schwerer Fehler) Der Linker hat nicht mehr genügend dynamisch zugewiesenen Speicher, der für den Linkvorgang erforderlich wäre. Normalerweise bedeutet das, daß zu viele Module, Externals, Gruppen oder Segmente in den gelinkten .OBJ-Dateien definiert sind. Sie können die Größe der RAM-Disks und/oder des Cache-Programms verringern. Unter Windows können Sie eine oder mehrere Anwendungen schließen, um Speicherplatz freizugeben.

out of memory creating extended dictionary
[Zu wenig Speicher für das Anlegen des erweiterten Verzeichnisses]
(Bibliotheksverwaltung, Fehler) Der Bibliotheksverwaltung steht nicht genügend Speicherplatz für das Anlegen des erweiterten Verzeichnisses zur Verfügung. Die Bibliothek wird angelegt, hat aber kein erweitertes Verzeichnis.

out of memory reading LE/LIDATA record from object module
[Zu wenig Speicher beim Lesen des LE/LIDATA-Records aus dem Objektmodul]
(Bibliotheksverwaltung, Fehler) Die Bibliotheksverwaltung hat versucht, einen Daten-Record aus dem Objektmodul zu lesen, aber es ist kein Speicherblock verfügbar, der groß genug ist. Wenn das hinzugefügte Modul ein großes Datensegment oder mehrere Segmente enthält, kann das Hinzufügen dieses Moduls

vor dem Einfügen anderer Module das Problem lösen. Wenn das Modul als erstes hinzugefügt wird, steht für das spätere Speichern von Modullisten und Listen öffentlicher Symbole genügend Speicherplatz zur Verfügung.

Out of space allocating per module debug struct
[Zu wenig Speicher für Modul-Debugger-Informationen]
(Bibliotheksverwaltung, Fehler) Die Bibliotheksverwaltung hat nicht genügend Speicherplatz für die Debugger-Informationen, die zu einem bestimmten Objektmodul gehören. Das Problem läßt sich evtl. durch Entfernen von Debugger-Informationen aus Modulen, die der Bibliothek hinzugefügt wurden, lösen.

Output device is full
[Ausgabegerät ist voll]
(Bibliotheksverwaltung, Fehler) Die Kapazität des Ausgabegeräts ist erschöpft. Normalerweise ist kein Platz mehr auf der Diskette/Platte.

Overload is now unnecessary and obsolete
[Das Schlüsselwort **overload** ist jetzt überflüssig]
(Compiler, Warnung) Frühere Versionen von C++ brauchten den Begriff **overload** für überladene Funktionsnamen. C++ verwendet jetzt »typensichere Bindung«, bei dem alle Funktionen als überladen angenommen werden, falls sie nicht anders markiert sind. **overload** sollte nicht mehr verwendet werden.

Overloadable operator expected
[Überladbarer Operator erwartet]
(Compiler, Fehler) Fast alle C++ Operatoren können überladen werden. Die einzigen nicht überladbaren sind der Punkt für die Feldauswahl `.`, der Punkt-Stern `.*`, der doppelte Doppelpunkt `::` und der bedingte Ausdruck `?:`. Die Operatoren # und ## des Präprozessors sind keine Operatoren der Sprachen C oder C++ und können deshalb nicht überladen werden. Interpunktionszeichen wie der Strichpunkt `;` sind keine Operatoren und können natürlich auch nicht überladen werden.

Overloaded *Funktionsname* ambiguous in this context
[Überladener *Funktionsname* ist in diesem Kontext mehrdeutig]
(Compiler, Fehler) Ein überladener Funktionsname kann nur dann ohne tatsächlichen Aufruf der Funktion verwendet werden, wenn eine Variable oder ein Parameter eines entsprechenden Typs damit initialisiert wird. Im vorliegenden Fall wurde ein überladener Funktionsname in einem anderen Kontext verwendet.

Overloaded prefix 'operator *Operator*' used as a postfix operator
[Der überladene Präfix-Operator *Operator* wurde als Suffix-Operator verwendet]
(Compiler, Warnung) In der aktuellen C++ Version kann sowohl die Präfix- als auch die Suffix-Version der Operatoren `++` und `--` überladen werden. Wenn älterer Programmtext, in dem nur der Präfix-Operator überladen ist, in einer Suffix-Umgebung eingesetzt wird, verwendet Turbo C++ den Präfix-Operator und gibt diese Warnung aus.

P1001 Unable to read file *Dateiname*
[Datei *Dateiname* kann nicht gelesen werden]
(Help-Projektdatei, Meldung) Die in der Projektdatei angegebene Datei kann nicht gelesen werden: DOS-Fehler.

P1003 Invalid path specified in Root option
[Ungültige Pfadangabe in der *Root*-Option]
(Help-Projektdatei, Meldung) Der in der *Root*-Option angegebene Pfad kann nicht gefunden werden. Der Compiler verwendet das aktuelle Arbeitsverzeichnis.

P1005 Path and filename exceed limit of 79 characters
[Pfad- und Dateiname länger als 79 Zeichen]
(Help-Projektdatei, Meldung) Der absolute Pfadname bzw. die Kombination aus Pfad zum Arbeitsverzeichnis und relativem Pfadnamen überschreitet die von DOS festgelegte Maximallänge von 79 Zeichen. Die Datei wurde übersprungen.

P1007 Root path exceeds maximum limit of 66 characters
[Root-Pfadname länger als 66 Zeichen]
(Help-Projektdatei, Meldung) Der Pfad zum angegebenen Arbeitsverzeichnis überschreitet die von DOS festgelegte Maximallänge von 66 Zeichen. Der Pfadname wird ignoriert und der Compiler verwendet das aktuelle Arbeitsverzeichnis.

P1009 [FILES] section missing
[Der [Files]-Abschnitt fehlt]
(Help-Projektdatei, Meldung) Der [Files]-Abschnitt ist obligatorisch. Die Compilierung wird abgebrochen.

P1011 Option *Optionsname* **previously defined**
[Option *Optionsname* bereits definiert]
(Help-Projektdatei, Meldung) Die angegebene Option wurde bereits definiert. Der Compiler ignoriert die neuerliche Definition.

P1013 Project file extension cannot be .HLP
[Die Namenserweiterung der Projektdatei ist unzulässig]
(Help-Projektdatei, Meldung) Sie können dem Compiler keine Projektdatei mit der Namenserweiterung .HLP angeben. Projektdateien haben normalerweise die Dateinamenserweiterung .HPJ.

P1015 Unexpected end-of-file
[Unerwartetes Dateiende]
(Help-Projektdatei, Meldung) Der Compiler ist unerwarteterweise an das Ende der Projektdatei gelangt. Vermutlich fehlt in einer einzubindenden Datei oder in der Projektdatei selbst ein abschließendes Kommentarzeichen.

P1017 Parameter exceeds maximum length of 128 characters
[Parameter länger als 128 Zeichen]
(Help-Projektdatei, Meldung) Eine Option, ein Kontextstring oder eine Kontext-

nummer, ein symbolischer Name oder ein anderer Parameter in der angegebenen Zeile ist länger als 128 Zeichen. Die Zeile wird ignoriert.

P1021 Context number already used in [MAP] section
[Kontextnummer bereits im [Map]-Abschnitt verwendet]
(Help-Projektdatei, Meldung) Die Kontextnummer in der angegebenen Zeile der Projektdatei wurde vorher bereits einem anderen Kontextstring zugeordnet. Die Zeile wird ignoriert.

P1023 Include statements nested too deeply
[Include-Anweisungen zu tief geschachtelt]
(Help-Projektdatei, Meldung) Die #*include*-Anweisung in der angegebenen Zeile hat die maximal zulässige Schachtelungstiefe von fünf Stufen überschritten.

P1025 Section heading *Abschnittname* unrecognized
[Unzulässiger *Abschnittname*]
(Help-Projektdatei, Meldung) Es wurde ein Abschnittkopf verwendet, der nicht vom Compiler unterstützt wird. Die Zeile wird übergangen.

P1027 Bracket missing from section heading *Abschnittname*
[Im Abschnitt *Abschnittname* fehlt die rechte eckige Klammer]
(Help-Projektdatei, Meldung) In dem angegebenen Abschnittkopf fehlt die rechte eckige Klammer (]). Fügen Sie diese ein und compilieren Sie erneut.

P1029 Section heading missing
[Abschnittkopf fehlt]
(Help-Projektdatei, Meldung) Der Abschnittkopf in der angegebenen Zeile ist nicht vollständig. Dieser Fehler wird auch gemeldet, wenn der erste Eintrag in einer Projektdatei kein Abschnittkopf ist. Der Compiler fährt mit der Bearbeitung der nächsten Zeile fort.

P1030 Section *Abschnittname* previously defined
[Abschnitt *Abschnittname* bereits definiert]
(Help-Projektdatei, Meldung) In der Projektdatei ist ein Abschnitt doppelt vorhanden. Die Zeilen unter dem wiederholten Abschnittkopf werden ignoriert und der Compiler setzt seine Bearbeitung bei dem nächsten gültigen Abschnitt fort.

P1031 Maximum number of build tags exceeded
[Zuviele symbolische Namen vorhanden]
(Help-Projektdatei, Meldung) Die maximale Anzahl für symbolische Namen beträgt 30. Die überzähligen Namen werden ignoriert.

P1033 Duplicate build tag in [BUILDTAGS] section
[Doppelter symbolischer Name im [BuildTags]-Abschnitt]
(Help-Projektdatei, Meldung) Im [BuildTags]-Abschnitt ist derselbe symbolische Name doppelt vorhanden.

P1035 Build tag length exceeds maximum
[Symbolischer Name länger als 32 Zeichen]
(Help-Projektdatei, Meldung) Der symbolische Name in der angegebenen Zeile ist länger als 32 Zeichen. Der Compiler übergeht diesen Eintrag.

P1037 Build tag *Name* contains invalid characters
[Der symbolische Name *Name* enthält unzulässige Zeichen]
(Help-Projektdatei, Meldung) Symbolische Namen dürfen nur alphanumerische Zeichen und den Unterstrich (_) enthalten. Die Zeile wird übergangen.

P1039 [BUILDTAGS] section missing
[[BuildTags]-Abschnitt fehlt]
(Help-Projektdatei, Meldung) Die Option *Build* deklarierte eine bedingte Compilierung, aber in der Projektdatei ist kein [BuildTags]-Abschnitt vorhanden. Alle Themen werden folglich compiliert.

P1043 Too many tags in Build expression
[Zu viele symbolische Namen im Build-Ausdruck]
(Help-Projektdatei, Meldung) Der Build-Ausdruck in der angegebenen Zeile enthält mehr als 20 symbolische Namen. Der Compiler ignoriert die Zeile.

P1045 [ALIAS] section found after [MAP] section
[[Alias]-Abschnitt steht nach [Map]-Abschnitt]
(Help-Projektdatei, Meldung) Der [Alias]-Abschnitt muß – wenn er verwendet wird – vor dem [Map]- Abschnitt stehen. Andernfalls wird er übergangen.

P1047 Context string *Kontextname* already assigned an alias
[Kontextstring *Kontextname* hat bereits einen Alias-Namen]
(Help-Projektdatei, Meldung) Folgende Zuweisung ist nicht möglich: a=b und dann a=c. Ein Kontextstring kann nur einen Alias-Namen haben. Der angegebene Kontextstring wurde im vorausgehenden [Alias]-Abschnitt zugewiesen. Die angeforderte Neuzuweisung in dieser Zeile wird ignoriert.

P1049 Alias string aliasname already assigned
[Alias-String hat bereits einen Alias-Namen]
(Help-Projektdatei, Meldung) Folgende Zuweisung ist nicht möglich: a=b und dann b=c. Sie können einem Alias-Namen keinen Alias-Namen zuweisen.

P1051 Context string *Kontextname* cannot be used as alias string
(Der Kontextstring *Kontextname* kann nicht als Alias-String verwendet werden)
(Help-Projektdatei, Meldung) Folgende Zuweisung ist nicht möglich: a=b und dann c=a. Ein Kontextstring, dem ein Aliasname zugeordnet wurde, kann später nicht als Aliasname für einen anderen Kontextstring verwendet werden.

P1053 Maximum number of font ranges exceeded
[Maximale Anzahl der Schriftgrößenbereiche überschritten]
Die maximale Anzahl für Schriftgrößenbereiche beträgt 5. Der Rest wird ignoriert.

P1055 Current font range overlaps previously defined range
[Aktueller Schriftgrößenbereich überlappt vorherigen Bereich]
Ein Schriftgrößenbereich überlappt eine zuvor durchgeführte Abbildung. Passen Sie die Schriftgrößenbereiche an, um Überlappungen zu entfernen. Die zweite Abbildung wird ignoriert.

P1056 Unrecognized font name in Forcefont option
[Unbekannter Schriftname in der *Forcefont*-Option]
(Help-Projektdatei, Meldung) Ein vom Compiler angetroffener Schriftname wird nicht unterstützt. Der Schriftname wird ignoriert und der Compiler verwendet die voreingestellte Helvetica-Schrift.

P1057 Font name too long
[Schriftname zu lang]
(Help-Projektdatei, Meldung) Der Schriftname darf nicht länger als 20 Zeichen ein. Die Schrift wird ignoriert.

P1059 Invalid multiple-key syntax
[Fehlerhafte Syntax in der *Multikey*-Option]
(Help-Projektdatei, Meldung) Die Syntax in der *Multikey*-Option ist nicht korrekt. Vergleichen Sie dazu den Abschnitt »Die Hilfe-Datei aufbauen« in Anhang A »HC: Der Hilfe-Compiler von Windows«.

P1061 Character already used
[Zeichen bereits verwendet]
(Help-Projektdatei, Meldung) Der Bezeichner für die zusätzliche Schlüsselworttabelle ist bereits vergeben. Wählen Sie ein anderes Zeichen.

P1063 Characters 'K' und 'k' cannot be used
['K' und 'k' dürfen nicht verwendet werden]
(Help-Projektdatei, Meldung) Diese beiden Zeichen sind für die reguläre Schlüsselworttabelle reserviert. Wählen Sie ein anderes Zeichen.

P1065 Maximum number of keyword tables exceeded
[Zu viele Schlüsselworttabellen vorhanden]
(Help-Projektdatei, Meldung) Es sind mehr als die zulässigen fünf Schlüsselworttabellen vorhanden. Die überzähligen Tabellen werden ignoriert.

P1067 Equal sign missing
[Gleichheitszeichen fehlt]
(Help-Projektdatei, Meldung) In einer Option in der angegebenen Zeile fehlt das erforderliche Gleichheitszeichen. Überprüfen Sie die Syntax dieser Option.

P1069 Context string missing
[Kontextstring fehlt]
(Help-Projektdatei, Meldung) In der angegebenen Zeile fehlt vor einem Gleichheitszeichen ein Kontextstring.

P1071 Incomplete line in *Abschnittname* section
[Unvollständige Zeile im Abschnitt *Abschnittname*]
(Help-Projektdatei, Meldung) Der Eintrag in der angegebenen Zeile ist nicht vollständig. Die Zeile wird übergangen.

P1073 Unrecognized option in [OPTIONS] section
[Unbekannte Option im [Options]-Abschnitt]
(Help-Projektdatei, Meldung) Es wurde eine vom Compiler nicht unterstützte Option verwendet. Die Zeile wird übergangen.

P1075 Invalid build expression
[Unzulässiger Build-Ausdruck]
(Help-Projektdatei, Meldung) Die Syntax im Build-Ausdruck in der angegebenen Zeile enthält einen oder mehrere logische oder syntaktische Fehler.

P1077 Warning level must be 1, 2, or 3
[Warning-Stufe muß 1, 2 oder 3 lauten]
(Help-Projektdatei, Meldung) Die *Warning*-Stufe kann auf 1, 2 oder 3 gesetzt werden. Gemäß Voreinstellung verwendet der Compiler die Stufe 3 (alle Warnungen).

P1079 Invalid compression option
[Fehlerhafte *Compress*-Option]
(Help-Projektdatei, Meldung) Die Option *Compress* kann nur auf *True* oder *False* gesetzt werden. Der Compiler führt seine Bearbeitung ohne Komprimierung fort.

P1081 Invalid title string
[Ungültiger Titelstring]
(Help-Projektdatei, Meldung) Die Option *Title* definiert einen String, der leer ist oder mehr als 32 Zeichen enthält. Im zweiten Fall wird der String gekürzt.

P1083 Invalid context identification number
[Ungültige Kontextnummer]
(Help-Projektdatei, Meldung) Die Kontextnummer in der angegebenen Zeile ist NULL oder enthält ungültige Zeichen.

P1085 Unrecognized text
[Unzulässiger Text]
(Help-Projektdatei, Meldung) Der unzulässige Text in der angegebenen Zeile, der auf korrekten Text folgt, wird ignoriert.

P1086 Invalid font-range syntax
[Fehlerhafte Syntax in der Schriftbereichdefinition]
(Help-Projektdatei, Meldung) Die Syntax der Schriftbereichdefinition in der angegebenen Zeile ist nicht korrekt. Der Compiler ignoriert die Zeile. Überprüfen Sie die Syntax in der *Mapfontsize*-Option.

P1089 Unrecognized sort ordering
[Unbekannte Sortierordnung]
(Help-Projektdatei, Meldung) Sie haben eine vom Compiler nicht unterstützte Sortierordnung angegeben. Wenden Sie sich an die Hotline der Borland GmbH.

Parameter names are used only with a function body
[Parameternamen brauchen einen Funktionsrumpf]
(Compiler, Fehler) Bei der Deklaration einer Funktion ohne Funktionsrumpf müssen entweder leere Klammern oder ein Funktionsprototyp angegeben werden. Eine Liste nur mit Parameternamen ist nicht erlaubt. Deklarationsbeispiele:

```
int func();            /* Deklaration ohne Prototyp - zulässig */
int func(int, int);    /* Deklaration mit Prototyp - zulässig */
int func(int i, int j); /* Parameternamen im Prototyp - zulässig */
int func(i, j);        /* Parameternamen allein - nicht zulässig */
```

Parameter *Nummer* missing name
[Parametername fehlt]
(Compiler, Fehler) Im Kopf der Funktionsdefinition stand hier nur eine Typangabe ohne Parameternamen. In C ist dies unzulässig, in C++ zwar erlaubt, es ist aber kein Zugriff auf den Parameter der Funktion möglich.

Parameter *Parameter* is never used
[Parameter *Parameter* wird nie verwendet]
(Compiler, Warnung) Ein bei der Definition angegebener Parameter wird innerhalb der Funktion nicht verwendet. Das kann, muß aber nicht, ein Fehler sein, der häufig durch einen fehlerhaft geschriebenen Parameter ausgelöst wird. Diese Meldung wird auch dann angezeigt, wenn der Bezeichner als automatische (lokale) Variable erneut im Funktionsrumpf deklariert wird. Der Parameter wird von der automatischen Variable überdeckt und nicht verwendet.

Pfad path is too long
[Pfad zu lang]
(Bibliotheksverwaltung, Fehler) Dieser Fehler tritt auf, wenn eine Bibliotheksdatei oder der Suchpfad einer Moduldatei mehr als 64 Zeichen enthält.

Pointer to structure required on left side of -> oder ->*
[Links vom Pfeil muß ein Zeiger stehen]
(Compiler, Fehler) Auf der linken Seite des Pfeils (->) darf in C oder C++ nur ein Zeiger stehen. In C++ trifft dies auch auf ->* zu.

Possible use of *Bezeichner* before definition
[Möglicherweise Verwendung von *Bezeichner* vor seiner Definition]
(Compiler, Warnung) Die in einem Ausdruck angegebene Variable wurde verwendet, bevor ihr ein Wert zugewiesen wurde. Der Compiler führt eine einfache Programmüberprüfung durch, um diese Fehlerquelle zu finden.

Wenn eine Variable physikalisch vor einer Zuweisung verwendet wird, erhalten Sie diese Meldung. Während des tatsächlichen Programmflusses kann der Wert dagegen vor der Verwendung der Variablen zugewiesen werden.

Possibly incorrect assignment
[Möglicherweise inkorrekte Zuweisung]
(Compiler, Warnung) Diese Warnung erhalten Sie, wenn der Compiler einen Zuweisungsoperator als Hauptoperator eines bedingten Ausdrucks antrifft (d.h. als Bestandteil einer **if**-, **while**- oder **do while**-Anweisung). Die Warnung wird häufig von einem typographischen Fehler im Gleicheitsoperator ausgelöst. Um die Warnung zu unterdrücken, schreiben Sie die Zuweisung in runde Klammern und vergleichen den gesamten Ausdruck explizit mit Null.

```
if (a = b) ...
```

sollte folgendermaßen geschrieben werden:

```
if ((a = b) != 0) ...
```

Public *Symbol* in modul *Modul1* clashes with prior module *Modul2*
[Öffentliches *Symbol* in *Modul1* überschneidet sich mit dem früheren Modul *Modul2*]
(Bibliotheksverwaltung, Fehler) Ein öffentliches Symbol darf in einer Bibliotheksdatei nur einmal auftreten. Ein zu der Bibliothek hinzugefügtes Modul enthält ein öffentliches *Symbol*, das bereits in einem Modul der Bibliothek vorhanden ist. Das neue Modul kann nicht hinzugefügt werden.

R2001 Unable to open bitmap file *Dateiname*
[Bitmap-Datei *Dateiname* kann nicht geöffnet werden]
(Help-Themendatei, Meldung) Die angegebene Bitmap-Datei kann nicht gelesen werden: DOS-Fehler.

R2003 Unable to include bitmap file *Dateiname*
[Bitmap-Datei *Dateiname* kann nicht aufgenommen werden]
(Help-Themendatei, Meldung) Die angegebene Bitmap-Datei konnte nicht gefunden oder gelesen werden. Es handelt sich entweder um einen DOS-Fehler oder es steht nicht genügend Speicherplatz zur Verfügung (»out of memory«).

R2005 Disk full
[Diskette/Platte voll]
(Help-Themendatei, Meldung) Die Ressourcendatei konnte nicht auf Platte/Diskette geschrieben werden. Schaffen Sie ausreichend Speicherplatz auf dem Ziellaufwerk.

R2009 Cannot use reserved DOS device name for file *Dateiname*
[Reservierter DOS-Gerätename kann nicht als Dateiname verwendet werden]
(Help-Themendatei, Meldung) Auf eine Datei wurde mit COM1, LPT2, PRN etc. Bezug genommen. Benennen Sie die Datei um.

R2013 Output file *Dateiname* already exists as a directory
[Ausgabedatei *Dateiname* existiert bereits als Verzeichnis]
(Help-Themendatei, Meldung) In dem Verzeichnis, das in der Projektdatei als

Arbeitsverzeichnis angegeben wurde, existiert ein Unterverzeichnis mit demselben Namen wie die gewünschte Ressourcendatei. Löschen Sie das Unterverzeichnis oder benennen Sie es um.

R2015 Output file *Dateiname* already exists as read-only
[Ausgabedatei *Dateiname* existiert bereits als Nur-Lesen-Datei]
(Help-Themendatei, Meldung) Der angegebene Dateiname kann von der Ressourcendatei nicht überschrieben werden, weil die Datei das Attribut Nur-Lesen hat. Benennen Sie die Projektdatei um oder ändern Sie das Attribut der Datei.

R2017 Path or file *Dateiname* exceeds limit of 79 characters
[Pfad- oder Dateiname *Dateiname* überschreitet die Länge von 79 Zeichen]
(Help-Themendatei, Meldung) Der absolute Pfadname bzw. die Kombination aus Pfad zum Arbeitsverzeichnis und Pfadname für die angegebene Datei überschreitet die von DOS zugelassene Länge von 79 Zeichen. Die Datei wird ignoriert.

R2019 Cannot open file *Dateiname*
[Datei *Dateiname* kann nicht geöffnet werden]
(Help-Themendatei, Meldung) Die angegebene Datei kann nicht gelesen werden: DOS-Fehler.

R2021 Cannot find file *Dateiname*
[Datei *Dateiname* kann nicht gefunden werden]
(Help-Themendatei, Meldung) Die angegebene Datei konnte nicht gefunden oder gelesen werden. Es handelt sich entweder um einen DOS-Fehler oder es steht nicht genügend Speicherplatz zur Verfügung (»out of memory«).

R2023 Not enough memory to build Help file
[Nicht genügend Speicherplatz für das Compilieren der Hilfedatei]
(Help-Themendatei, Meldung) Schaffen Sie freien Speicherplatz: Entfernen Sie nicht benötigte Anwendungen, Gerätetreiber und speicherresidente Programme aus dem Arbeitsspeicher.

R2025 File environment error
[Datei-Umgebungsfehler]
(Help-Themendatei, Meldung) Der Compiler hat nicht genügend Datei-Handles für die Weiterarbeit zur Verfügung. Erhöhen Sie die Werte für FILES und BUFFERS in der Datei CONFIG.SYS und booten Sie erneut.

R2027 Build tag *Name* not defined in [BUILDTAGS] section of project file
[Der symbolische Name *Name* ist im [BuildTags]-Abschnitt der Projektdatei nicht definiert]
(Help-Themendatei, Meldung) Der angegebene symbolische Name wurde zwar einem Thema zugewiesen, aber in der Projektdatei nicht deklariert. Der Name wird für das Thema ignoriert.

R2033 Context string in [MAP] section not defined in any topic
[Kontextstring im [Map]-Abschnitt in keinem Thema definiert]
(Help-Themendatei, Meldung) In der Projektdatei sind ein oder mehrere Kontextstrings definiert, der Compiler kann aber keine entsprechenden Themen finden.

R2035 Build expression missing from project file
[Build-Ausdruck nicht in der Projektdatei vorhanden]
(Help-Themendatei, Meldung) Die Themen haben symbolische Namen, aber es gibt keinen Build-Ausdruck in der Projektdatei. Der Compiler schließt alle Themen in die Compilierung ein.

R2037 File *Dateiname* cannot be created, due to previous error(s)
[Datei *Dateiname* kann aufgrund vorausgehender Fehler nicht erstellt werden]
(Help-Themendatei, Meldung) Die Ressorucendatei kann nicht erstellt werden, weil der Compiler keine Themen gefunden hat, die bearbeitet werden können. Korrigieren Sie diesem Fehler vorausgehende Fehler und compilieren Sie danach erneut.

R2039 Unrecognized table formatting in topic *Themennummer* of file *Dateiname*
[Unbekannte Tabellenformatierung in Thema *Themennummer* in der Datei *Dateiname*]
(Help-Themendatei, Meldung) Der Compiler ignoriert die Tabellenformatierung, die vom Hilfesystem nicht unterstützt wird. Formatieren Sie die Einträge als fortlaufenden Text neu, falls dies möglich ist.

R2041 Jump *Kontext_String* unresolved in topic *Themennummer* of file *Dateiname*
[Sprung-Kontextbegriff im Thema *Themennummer* in der Datei *Dateiname* nicht auswertbar]
(Help-Themendatei, Meldung) Das angegebene Thema enthält einen Kontextstring, der ein nicht vorhandenes Thema kennzeichnet. Prüfen Sie auf Tippfehler, damit das gewünschte Thema in die Compilierung eingeschlossen wird.

R2043 Hotspot text cannot spread over paragraphs
[Der Sprungbegriff darf sich nicht auf mehrere Absätze beziehen]
(Help-Themendatei, Meldung) Ein Sprungbegriff erstreckt sich auf zwei Absätze. Entfernen Sie die Absatzmarkierung oder ändern Sie die Formatierung des Sprungbegriffs.

R2045 Maximum number of tab stops reached in topic *Themennummer* of file *Dateiname*
[Maximale Anzahl von Tabulatoren im Thema *Themennummer* der Datei *Dateiname* erreicht]
(Help-Themendatei, Meldung) Im angegebenen Thema sind mehr als die 32

zulässigen Tabulatoren vorhanden. Nach der 32. Tabulatorstelle wird die voreingestellte Tabulierung verwendet.

R2047 File *Dateiname* not created
[Datei *Dateiname* nicht angelegt]
(Help-Themendatei, Meldung) Es sind keine Themen für die Compilierung vorhanden oder der Build-Ausdruck ist für alle Themen *false*. Es wird keine Ressourcendatei erstellt.

R2049 Context string text too long in topic *Themennummer* of file *Dateiname*
[Der Kontextstring im Thema *Themennummer* der Datei *Dateiname* ist zu lang]
(Help-Themendatei, Meldung) Der verborgene Kontextstring darf nicht länger als 64 Zeichen sein. Der String wird ignoriert.

R2051 File *Dateiname* is not a valid RTF topic file
[Die Datei *Dateiname* ist keine gültige RTF-Themendatei]
(Help-Themendatei, Meldung) Die angegebene Datei ist keine gültige RTF-Datei. Überprüfen Sie, ob Sie die Themen mit Ihrem Textprogramm im RTF-Format abgespeichert haben.

R2053 Font *Schriftname* in file *Dateiname* not in RTF font table
[Die Schrift *Schriftname* in der Datei *Dateiname* befindet sich nicht in der RTF-Schriftentabelle]
(Help-Themendatei, Meldung) Das Thema wurde in einer Schrift erfaßt, die nicht im RTF-Kopf definiert ist. Der Compiler verwendet die voreingestellte Schrift.

R2055 File *Dateiname* is not a usable RTF topic file.
[Die Datei *Dateiname* ist keine verwendbare RTF-Themendatei]
(Help-Themendatei, Meldung) Die angegebene Datei enthält einen gültigen RTF-Kopf, der Inhalt entspricht aber nicht oder nur teilweise dem RTF-Format.

R2057 Unrecognized graphic format in topic *Themennummer* of file *Dateiname*
[Unbekanntes Grafikformat im Thema *Themennummer* der Datei *Dateiname*]
(Help-Themendatei, Meldung) Der Compiler unterstützt nur Windows-Bitmaps. Stellen Sie sicher, daß keine Meta-Dateien oder Macintosh-Formate verwendet wurden. Die Grafik wird ignoriert.

R2059 Context string identifier already defined in topic *Themennummer* of file *Dateiname*
[Der Kontextstring-Bezeichner ist bereits im Thema *Themennummer* der Datei *Dateiname* definiert]
(Help-Themendatei, Meldung) Es gibt mehr als eine Bezeichner-Fußnote für den Kontextstring des angegebenen Themas. Der Compiler verwendet den in der ersten Fußnote (#) definierten Bezeichner.

R2061 Context string *Kontextname* already used in file *Dateiname*
[Der Kontextstring *Kontextname* ist schon in der Datei *Dateiname* vorhanden]
Der angegebene Kontextstring wurde zuvor bereits einem Thema zugeordnet. Der Compiler ignoriert den folgenden String, und das Thema hat damit keinen Bezeichner.

R2063 Invalid context-string identifier for topic *Themennummer* of file *Dateiname*
[Unzulässiger Kontextstring-Bezeichner für das Thema *Themennummer* der Datei *Dateiname*]
(Help-Themendatei, Meldung) Die Kontextstring-Fußnote enthält Zeichen, die nicht alphanumerisch sind, oder ist leer. Dem Thema wird kein Bezeichner zugewiesen.

R2065 Context string defined for index topic is unresolved
[Für das Indexthema definierter Kontextstring kann nicht gefunden werden]
(Help-Themendatei, Meldung) Das Indexthema in der Projektdatei kann nicht gefunden werden. Der Compiler verwendet dafür das erste Thema.

R2067 Footnote text too long in topic *Themennummer* of file *Dateiname*
[Der Text der Fußnote im Thema *Themennummer* der Datei *Dateiname* ist zu lang]
(Help-Themendatei, Meldung) Der Text einer Fußnote darf nicht mehr als 1000 Zeichen umfassen. Die Fußnote wird ignoriert.

R2069 Build tag footnote not at the beginning of topic *Themennummer* of file *Dateiname*
[Die Fußnote des symbolischen Namens steht nicht am Beginn des Themas *Themennummer* der Datei *Dateiname*]
(Help-Themendatei, Meldung) Das angegebene Thema enthält eine Fußnote für einen symbolischen Namen, die nicht als erstes Zeichen im Thema steht. Dem Thema wird kein symbolischer Name zugewiesen.

R2071 Footnote text missing in topic *Themennummer* of file *Dateiname*
[Im Thema *Themennummer* der Datei *Dateiname* fehlt der Fußnotentext]
(Help-Themendatei, Meldung) Das angegebene Thema enthält eine »leere« Fußnote.

R2073 Keyword string is null in topic *Themennummer* of file *Dateiname*
[Der Schlüsselwortstring im Thema *Themennummer* der Datei *Dateiname* ist leer]
(Help-Themendatei, Meldung) Für das angegebene Thema existiert eine Schlüsselwortfußnote, die aber keine Zeichen enthält.

R2075 Keyword string too long in topic *Themennummer* of file *Dateiname*
[Im Thema *Themennummer* der Datei *Dateiname* ist das Schlüsselwort zu lang]
(Help-Themendatei, Meldung) Der Text in der Schlüsselwortfußnote des angegebenen Themas umfaßt mehr als die 255 zugelassenen Zeichen. Die überzähligen Zeichen werden ignoriert.

R2077 Keyword(s) defined without title in topic *Themennummer* of file *Dateiname*
[Im Thema *Themennummer* der Datei *Dateiname* haben die Schlüsselwörter keinen Titel]
(Help-Themendatei, Meldung) Für das angegebene Thema wurde ein Schlüsselwort (bzw. Schlüsselwörter) definiert, aber dem Thema wurde kein Titel zugewiesen.

In der Liste der gefundenen Themen wird dieses Thema als Thema ohne Titel aufgeführt.

R2079 Browse sequence string is null in topic *Themennummer* of file *Dateiname*
[Der String der Suchsequenz im Thema *Themennummer* der Datei *Dateiname* ist leer]
Die Suchsequenzfußnote für das angegebene Thema enthält keine Sequenz-Zeichen.

R2081 Browse sequence string too long in topic *Themennummer* of file *Dateiname*
[Der String der Suchsequenz im Thema *Themennummer* der Datei *Dateiname* ist zu lang]
(Help-Themendatei, Meldung) Die Suchsequenzfußnote für das angegebene Thema umfaßt mehr als die zulässigen 128 Zeichen. Die Sequenz wird für das Thema ignoriert.

R2083 Missing sequence number in topic *Themennummer* of file *Dateiname*
[Im Thema *Themennummer* der Datei *Dateiname* fehlt die Sequenznummer]
(Help-Themendatei, Meldung) Die Suchsequenz für das angegebene Thema endet mit einem Doppelpunkt (:). Löschen Sie den Doppelpunkt oder geben Sie eine »kleine« Sequenznummer ein.

R2085 Sequence number already defined in topic *Themennummer* of file *Dateiname*
[Für das Thema *Themennummer* der Datei *Dateiname* existiert bereits eine Sequenznummer]
(Help-Themendatei, Meldung) Für das angegebene Thema existiert bereits eine Suchsequenzfußnote. Die folgende Sequenz wird ignoriert.

R2087 Build tag too long
[Der symbolische Name ist zu lang]
(Help-Themendatei, Meldung) Der symbolische Name für das angegebene Thema umfaßt mehr als die zulässigen 32 Zeichen. Der Name wird für das Thema ignoriert.

R2089 Title string null in topic ***Themennummer*** **of file** ***Dateiname***
[Der Titelstring im Thema *Themennummer* der Datei *Dateiname* ist leer]
(Help-Themendatei, Meldung) Die Titelfußnote für das angegebene Thema enthält keine Zeichen. Dem Thema wird kein Titel zugewiesen.

R2091 Title too long in topic ***Themennummer*** **of file** ***Dateiname***
[Der Titel im Thema *Themennummer* der Datei *Dateiname* ist zu lang]
(Help-Themendatei, Meldung) Der Titel für das angegebene Thema umfaßt mehr als die zulässigen 128 Zeichen. Diese überzähligen Zeichen werden ignoriert.

R2093 Title titlename in topic ***Themennummer*** **of file** ***Dateiname*** **used previously**
[Der Titel im Thema *Themennummer* der Datei *Dateiname* wurde bereits verwendet]
(Help-Themendatei, Meldung) Der angegebene Titel wurde bereits einem anderen Thema zugewiesen.

R2095 Title defined more than once in topic ***Themennummer*** **of file** ***Dateiname***
[Dem Thema *Themennummer* der Datei *Dateiname* wurden mehrere Titel zugewiesen]
(Help-Themendatei, Meldung) Für das angegebene Thema ist mehr als ein Titel vorhanden. Der Compiler verwendet den ersten Titelstring.

R2501 Using old key-phrase table
[Veraltete Schlüsselsatztabelle]
(Help-Themendatei, Meldung) Eine maximale Komprimierung kann nur erreicht werden, wenn vor der erneuten Compilierung der Hilfethemen die Datei .PH gelöscht wird.

R2503 Out of memory during text compression
[Zu wenig Arbeitsspeicher für die Textkomprimierung]
(Help-Themendatei, Meldung) Der Compiler hat zur Komprimierung nicht genügend Arbeitsspeicher. Die Compilierung der Ressourcendatei wird ohne Komprimierung fortgesetzt. Entfernen Sie nicht benötigte Anwendungen, Gerätetreiber und speicherresidente Programme.

R2505 File environment error during text compression
[Nicht genügend Datei-Handles für die Komprimierung vorhanden]
(Help-Themendatei, Meldung) Dem Compiler stehen nicht genügend Datei-

Handles für die Komprimierung zur Verfügung. Die Compilierung der Ressourcendatei wird ohne Komprimierung fortgesetzt. Erhöhen Sie die Werte für FILES =und BUFFERS =in Ihrer CONFIG.SYS-Datei und booten Sie Ihr System.

R2507 DOS file error during text compression
[DOS-Dateifehler während der Komprimierung]
(Help-Themendatei, Meldung) Der Compiler hat während der Komprimierung Schwierigkeiten beim Zugriff auf eine Datei. Die Compilierung der Ressourcendatei wird ohne Komprimierung fortgesetzt.

R2509 Error during text compression
[Fehler während der Komprimierung]
(Help-Themendatei, Meldung) Es ist ein Komprimierungsfehler aufgetreten: R2503, R2505 oder R2507. Die Compilierung der Ressourcendatei wird ohne Komprimierung fortgesetzt.

R2701 Internal error.
R2703 Internal error.
R2705 Internal error.
R2707 Internal error.
R2709 Internal error.
[Interne Fehler]
Wenden Sie sich an die Hotline der Borland GmbH.

record kind *Num* **found, expected theadr or lheadr in module** *Dateiname*
[Record-Typ *Num* gefunden, im Modul *Dateiname* wird *theadr* oder *lheadr* erwartet]
(Bibliotheksverwaltung, Fehler) Die Bibliotheksverwaltung konnte den Kopf-Record des Objektmoduls, das der Bibliothek hinzugefügt wurde, nicht interpretieren, und geht deshalb davon aus, daß es sich um ein unzulässiges Modul handelt.

record length *Länge* **exceeds available buffer in module** *Modul*)
[Die Record-Länge *Länge* ist für den verfügbaren Puffer in Modul *Modul* zu groß]
(Bibliotheksverwaltung, Fehler) Dieser Fehler tritt auf, wenn die Record-Länge *Länge* für den verfügbaren Puffer des Moduls *Modul* zu groß ist. Dies ist der Fall, wenn die Bibliotheksverwaltung nicht genügend Speicher hat.

record type *Typ* **found, expected theadr or lheadr in** *Modul*
[Record-Typ *Typ* gefunden, in *Modul* wurde *theadr* oder *lheadr* erwartet]
(Bibliotheksverwaltung, Fehler) Die Bibliotheksverwaltung hat im Modul *Modul* anstelle von *theadr* oder *lheadr* einen nicht erwarteten Typ *Typ* gefunden.

Redefinition of *Makro* **is not identical**
[Nicht identische Neudefinition des Makros *Makro*]
(Compiler, Warnung) Ein bereits definiertes Makro wurde durch eine erneute

Definition verändert. Der Text entspricht jedoch nicht exakt der ersten Definition. Der neue Text ersetzt die alte Definition.

Reference initialized with ***Typ1*****, needs lvalue of type** ***Typ2***
[Die mit *Typ1* initialisierte Referenz benötigt einen **lvalue** vom *Typ2*]
(Compiler, Fehler) Eine Referenzvariable oder ein Referenzparameter, die bzw. der nicht vom Typ **const** ist, muß mit einem entsprechenden **lvalue**-Typ initialisiert werden. Im vorliegenden Fehlerfall war der Initialisierer entweder kein **lvalue** oder sein Typ paßt nicht zu der zu initialisierenden Referenz.

Reference member ***Element*** **in class without constructors**
[Referenzelement *Element* steht in einer Klasse ohne Konstruktoren]
(Compiler, Fehler) Eine Klasse, die Referenzelemente enthält, muß mindestens einen benutzerdefinierten Konstruktor aufweisen, andernfalls können derartige Elemente nicht initialisiert werden.

Reference member ***Element*** **is not initialized**
[Referenzelement *Element* ist nicht initialisiert]
(Compiler, Fehler) Referenzen müssen immer initialisiert werden. Ein Klassenelement vom Referenztyp braucht einen Initialisierer in allen Konstruktoren dieser Klasse. Sie können nicht davon ausgehen, daß der Compiler Konstruktoren für solche Klassen generiert, da die notwendigen Daten für die Initialisierung der Referenzen fehlen.

Reference member ***Element*** **needs a temporary for initialization**
[Referenzelement *Element* benötigt temporäre Größe zur Initialisierung]
(Compiler, Fehler) Der Benutzer lieferte einen Anfangswert für einen Referenztyp, der kein **lvalue** des referenzierten Typs war. Der Compiler erzeugt dann eine temporäre Größe für die Initialisierung. Weil es keinen offensichtlichen Platz für diese temporäre Größe gibt, ist diese Initialisierung unzulässig.

Reference variable ***Variable*** **must be initialized**
[Referenzvariable *Variable* muß initialisiert sein]
(Compiler, Fehler) Dieses C++ Objekt ist als Referenz deklariert, aber nicht initialisiert. Alle Referenzen müssen zum Zeitpunkt ihrer Deklaration initialisiert sein.

Register allocation failure
[Fehler bei der Registerzuweisung]
(Compiler, schwerer Fehler) Diese Meldung signalisiert einen Compilerfehler. Ein Ausdruck in der angegebenen Zeile ist so komplex, daß der Code-Generator dafür keinen Code erzeugen kann. Vereinfachen Sie den Ausdruck. Wenn auch eine Unterteilung des Ausdrucks in mehrere Einzelbefehle nicht weiterhilft, kontaktieren Sie bitte die Hotline der Borland GmbH.

Results are safe in file ***Dateiname***
[Die Ergebnisse sind in der Datei *Dateiname* gesichert]
(Bibliotheksverwaltung, Warnung) Die Bibliotheksverwaltung hat die Biblio-

thek erfolgreich in einer temporären Datei aufgebaut. Es ist aber nicht möglich, die Datei mit dem gewünschten Dateinamen zu benennen. Die temporäre Datei wird nicht gelöscht, die Bibliothek also »gerettet«.

Size of *Bezeichner* is unknown or zero
[Die Größe von *Bezeichner* ist unbekannt oder Null]
(Compiler, Fehler) Dieser Bezeichner wurde in einem Zusammenhang verwendet, in dem seine Größe gebraucht wurde. Ein **struct**-»Tag« (ohne Definition des **struct**) oder ein **extern**-Array darf ohne Größe deklariert werden. Dann ist es aber nicht zulässig, sich auf eine derartige Konstruktion zu beziehen (z.B. mit **sizeof**) oder einen Zeiger auf diesen Typ zu dereferenzieren.
Verändern Sie die Deklaration so, daß die benötigte Größe von *Bezeichner* verfügbar ist.

sizeof may not be applied to a bit field
[**sizeof** darf nicht auf ein Bitfeld angewandt werden]
(Compiler, Fehler) **sizeof** liefert die Größe eines Datenobjekts in Bytes; dies ist für ein Bitfeld nicht möglich.

sizeof may not be applied to a function
[**sizeof** darf nicht auf eine Funktion angewandt werden]
(Compiler, Fehler) **sizeof** darf nur auf Datenobjekte, nicht auf Funktionen angewandt werden. Die Größe eines Zeigers auf eine Funktion können Sie aber mit **sizeof** ermitteln.

Size of the type is unknown or zero
[Größe des Typs unbekannt oder Null]
(Compiler, Fehler) Dieser Typ wurde in einem Zusammenhang verwendet, in dem seine Größe gebraucht wurde. Ein **struct**-»Tag« kann z.B. (ohne Definition des **struct**) ohne Größe deklariert werden. Dann ist es aber nicht zulässig, sich auf eine derartige Konstruktion zu beziehen (z.B. mit **sizeof**) oder einen Zeiger auf diesen Typ zu dereferenzieren. Verändern Sie die Deklaration so, daß die benötigte Größe verfügbar ist.

***Bezeichner* specifies multiple or duplicate access**
[*Bezeichner* bestimmt uneinheitliche Zugriffsrechte]
(Compiler, Fehler) Eine Basisklasse kann entweder **public** oder **private** deklariert sein, aber nicht beides. Der Zugriffsspezifizierer darf für eine Basisklasse nur einmal erscheinen.

Stack overflow
[Stack-Überlauf]
(Laufzeit, Fehler) Die voreingestellte Größe des Stack von Turbo C++ Programmen ist 5120 Bytes. Dies ist für die meisten Programme ausreichend. Programme mit rekursiven Funktionen oder mit Funktionen, die viele lokale Daten speichern, könnten aber den Stack zum Überlauf bringen. Diese Meldung erhalten Sie nur, wenn Sie die Option zur Überprüfung des Stack eingeschaltet haben. Vergrößern Sie den Stack oder reduzieren Sie das Ausmaß, in dem Ihr Programm

vom Stack abhängig ist. Unter Windows verwenden Sie STACKSIZE in Ihrer Moduldefinitionsdatei.

Das folgende Beispiel demonstriert, wie Sie den Speicherverbrauch von lokalen Variablen reduzieren können. Die Variable *buffer* wurde als statische Variable deklariert und verbraucht keinen Stack-Speicherplatz, wie es bei der Variablen *list* der Fall ist:

```
void anyfunction(void)
{
   static int buffer[2000]; /* im Datensegment */
   int list[2000];          /* auf dem Stack */
   }
```

Die Deklaration lokaler Variablen als **static** hat auch Nachteile:

1. Statische lokale Variablen reduzieren den Heap und den Speicher, der für globale Variablen zur Verfügung steht. Normalerweise kann man das in Kauf nehmen.

2. Die Funktion könnte dadurch nicht mehr wiedereintrittsfähig sein. Wenn die Funktion nämlich rekursiv oder asynchron aufgerufen wird und jeder Aufruf davon abhängt, daß er über eine eigene eindeutige Kopie einer bestimmten Variablen verfügt, dann können Sie diese Variable nicht als statisch deklarieren. In diesem Fall würde jeder Aufruf mit exakt demselben Speicherbereich arbeiten und das Programm könnte völlig unerwartete oder, bei Netz- oder Multitasking-Anwendungen, unvorhersehbare Ergebnisse produzieren, wenn die Funktion sich selbst auszuführen versucht (rekursiv) oder von verschiedenen Programmteilen zur selben Zeit aufgerufen wird (asynchron). Bei den meisten DOS-Programmen ist dies jedoch kein Problem.

Stack size is less than 1400h. It has been reset to 1400h
[Die Stack-Größe liegt unter 1400h sie wurde auf 1400h gesetzt]
(Linker, Warnung) Windows 3.0 benötigt für eine Anwendung eine Stack-Größe von mindestens 1400h. Wenn das automatische Datensegment (ADS) fast 64 KByte groß, der Stack aber kleiner als 1400h ist, kann das Datensegment beim Laden allerdings nicht beim Linken überlaufen. Um dies bei einer Windows-Anwendung zu verhindern, erzwingt der Linker eine Stack-Größe von mindestens 1400h.

Statement missing ;
[Abschließendes ; fehlt]
(Compiler, Fehler) Der Compiler hat eine Ausdrucksanweisung gefunden, nach der kein Strichpunkt steht.

Storage class *Speicherklasse* is not allowed here
[Speicherklasse *Speicherklasse* ist hier nicht erlaubt]
(Compiler, Fehler) Die angegebene Speicherklasse ist hier nicht erlaubt. Vermutlich wurden zwei Speicherklassen angegeben, obwohl nur eine angegeben werden darf.

Structure passed by value
[Struktur als Wert übergeben]
(Compiler, Warnung) Eine Struktur wurde über ihren Wert als Argument an eine Funktion ohne Prototyp übergeben. In den meisten Fällen dürfte es sich um einen vergessenen Adreß-Operator (&) handeln. Da Strukturen über ihren Wert übergeben werden können, kann das Fehlen des Operators akzeptiert werden. Die Meldung soll Sie nur auf diesen Fehler aufmerksam machen.

Structure required on left side of . or .*
[Auf der linken Seite von . oder .* muß eine Struktur stehen]
(Compiler, Fehler) Die linke Seite eines Punkt-Operators (in C++ auch eines Punkt-Sternchen-Operators) muß als Strukturtyp ausgewertet werden – was nicht der Fall war.

Structure size too large
[Struktur zu groß]
(Compiler, Fehler) Ihre Quelldatei enthält die Deklaration einer Struktur mit mehr als 64 KByte.

Stub program exceeds 64K
[Stub-Programm größer als 64 KByte]
(Linker, schwerer Fehler) Ein für eine Windows-Anwendung geschriebenes DOS-Stub-Programm umfaßt mehr als 64 KByte. Stub-Programme werden mit Hilfe der Anweisung STUB in der Moduldefinitionsdatei angegeben. Der Linker unterstützt Stub-Programme nur bis zu einer Größe von 64 KByte.

Style of function definition is now obsolete
[Dieser Stil der Funktionsdefinition ist veraltet]
(Compiler, Warnung) In C++ ist dieser Stil einer C-Funktionsdefinition nicht mehr zulässig:

```
int func(p1, p2)
int p1, p2;
{
.
.
}
```

Auch andere C++Compiler erlauben dies möglicherweise nicht.

Subscripting missing]
[Array-Indizierung: abschließendes] fehlt]
(Compiler, Fehler) Dieser Fehler kann auch durch nicht abgeschlossene Klammerebenen und überzählige Operatoren ausgelöst werden.

Superfluous & with function
[Überflüssiger Adreß-Operator bei Funktion]
(Compiler, Warnung) Der Adreß-Operator & ist bei einem Funktionsnamen nicht notwendig; derartige Operatoren werden verworfen.

Suspicious pointer conversion
[Möglicherweise unkorrekte Zeigerumwandlung]
(Compiler, Warnung) Der Compiler hat eine Zeigerumwandlung gefunden, die den Zeiger auf einen unterschiedlichen Typ zeigen läßt. Sie sollten eine Typumwandlung durchführen, um die Warnung zu unterdrücken, wenn die Umwandlung tatsächlich richtig ist.

Switch selection expression must be of integral type
[In **switch**-Auswahl nur ganzzahlige Werte]
(Compiler, Fehler) Der Auswahlausdruck in den Klammern einer **switch**-Anweisung muß ganzzahlig sein (**char**, **short**, **int**, **long**, **enum**). Vielleicht ist dies mit einer expliziten Typumwandlung zu erreichen.

Switch statement missing (
[In der **switch**-Anweisung fehlt ( ]
(Compiler, Fehler) Nach dem Schlüsselwort **switch** fehlt in der **switch**-Anweisung eine öffnende Klammer.

Switch statement missing)
[In der **switch**-Anweisung fehlt)]
(Compiler, Fehler) Nach dem zu prüfenden Ausdruck fehlt in der **switch**-Anweisung eine schließende Klammer.

***Dateiname* (*Zeilennummer*): Syntax error**
Dateiname (*Zeilennummer*): (Syntaxfehler]
(Linker, schwerer Fehler) Der Linker hat in der Moduldefinitionsdatei einen Syntaxfehler gefunden. Über *Dateiname* und *Zeilennummer* wird Ihnen die genaue Position des Fehlers mitgeteilt.

Table limit exceeded
[Überlauf einer internen Tabelle]
(Linker, schwerer Fehler) Es ist zu einem Überlauf in einer internen Linker-Tabelle gekommen. Das bedeutet, daß die gelinkten Programme die Kapazität des Linkers für die Aufnahme von öffentlichen und externen Symbolen und logischen Segmentdefinitionen erschöpft haben. Jede Instanz eines unterschiedlichen Segmentnamens in einer .OBJ-Datei zählt als ein logisches Segment; wenn zwei .OBJ-Dateien dieses Segment definieren, ergibt dies zwei logische Segmente.

Template argument must be a constant expression
[Template-Argument muß ein konstanter Ausdruck sein]
(Compiler, Fehler) Ein nicht-typisiertes Argument einer Template-Klasse muß ein konstanter Ausdruck entsprechenden Typs sein. Dazu gehören konstante Integerausdrücke und Adressen von Objekten oder Funktionen mit externer Bindung oder Elemente.

Template class nesting too deep: 'class'
[Template-Klasse zu tief geschachtelt: 'Klasse']
(Compiler, Fehler) Der Compiler schreibt eine bestimmte Begrenzung für die Anzahl Schachtelungsstufen bei Template-Klassen vor. Diese Grenze wird normalerweise nur durch eine rekursive Abhängigkeit von Template-Klassen überschritten. Wenn die Schachtelungstiefe überschritten wird, gibt der Compiler diese Fehlermeldung für alle geschachtelten Template-Klassen aus. Dadurch läßt sich die Rekursion einfach feststellen. Dieser Fehler zieht einen schweren »Out of memory«-Fehler nach sich.
Sehen Sie sich folgende Template-Klassen an:

```
template<class T> class A
{
   friend class B<T*>;
};

template<class T> class B
{
   friend class A<T>;
};

A<int> x;
```

Die obige Auflistung würde folgende Fehlermeldungen nach sich ziehen:

```
Error: Template class nesting too deep: 'B<int*****>'
Error: Template class nesting too deep: 'A<int****>'
Error: Template class nesting too deep: 'B<int****>'
Error: Template class nesting too deep: 'A<int***>'
Error: Template class nesting too deep: 'B<int***>'
Error: Template class nesting too deep: 'A<int**>'
Error: Template class nesting too deep: 'B<int**>'
Error: Template class nesting too deep: 'A<int*>'
Error: Template class nesting too deep: 'B<int*>'
Error: Template class nesting too deep: 'A<int>'
Fatal: Out of memory
```

Template function argument *Argument* **not used in argument types**
[Das Argument *Argument* der Template-Funktion befindet sich nicht unter den Argumenttypen]
(Compiler, Fehler) Das gegebene Argument ist nicht in der Argumentliste der Funktion vorhanden. Die Argumentliste einer Template-Funktion muß alle formalen Argumente der Schablone umfassen, andernfalls kann keine Instanz der Template-Funktion basierend auf den Argumenttyen erzeugt werden.

Template functions may only have type-arguments
[Template-Funktionen können nur **type**-Argumente haben]
(Compiler, Fehler) Ein Funktions-Template wurde mit einem nicht-**type**-Argument deklariert. Dies ist bei Template-Funktionen nicht zulässig, da bei ihrem Aufruf kein Wert festgelegt werden kann.

Templates can only be declared at file level
[Templates können nur auf Dateiebene deklariert werden]
(Compiler, Fehler) Templates können nicht innerhalb von Klassen oder Funktionen deklariert werden. Sie müssen globalen Gültigkeitsbereich haben und daher auf Dateiebene deklariert werden.

Templates must be classes or functions
[Templates müssen Klassen oder Funktionen sein]
(Compiler, Fehler) Die Deklaration eines Template muß entweder einen Klassentyp oder eine Funktion angeben.

Temporary used to initialize *Bezeichner*
Temporary used for parameter *Nummer* **in call to** *Funktion*
Temporary used for parameter *Parameter* **in call to** *Funktion*
Temporary used for parameter *Nummer*
Temporary used for parameter *Parameter*
[Verwendung temporärer Größen]
(Compiler, Warnungen) In C++ müssen sich Referenzvariablen oder -parameter auf ein Objekt desselben Typs beziehen. Stimmen die Typen nicht überein, wird der Wert einer temporären Größe desselben Typs zugewiesen; die Adresse der temporären Größe wird der Referenzvariablen oder dem Referenzparameter zugewiesen.

Die Warnung zeigt, daß die Referenz nicht auf das Erwartete, sondern auf eine temporäre Variable verweist, die sonst nicht benutzt wird.

Z.B. braucht eine Funktion *f* eine Referenz auf einen **int**, *c* ist aber **char**:

```
f(int&);
char c;
f(c);
```

Anstatt *f* mit der Adresse von *c* aufzurufen, generiert der Compiler Code, der folgendem C++ Quelltext entspricht:

```
int X = c, f(X);
```

Terminated by user
[Abbruch durch den Anwender]
(Linker, schwerer Fehler) Sie haben den Linkvorgang abgebrochen.

The constructor *Konstruktor* **is not allowed**
[Konstruktor *Konstruktor* hier nicht erlaubt]
(Compiler, Fehler) Konstruktoren der Form *X::(X)* sind nicht erlaubt. Die richtige Schreibweise für einen Kopierkonstruktor ist: *X::(const X&)*.

The value for *Bezeichner* **is not within the range of an int**
[Der Wert von *Bezeichner* ist kein **int**-Wert]
(Compiler, Fehler) Alle Aufzählungswerte müssen als Integerwert dargestellt werden können. Sie haben versucht, einen Wert zuzuweisen, der nicht vom Typ **int** ist. Wenn Sie in C++ eine Konstante für diesen Wert benötigen, verwenden Sie einen geeigneten Integertyp mit **const**.

'this' can only be used within a member function
[**this** kann nur in einer Elementfunktion verwendet werden]
(Compiler, Fehler) In C++ ist **this** ein reserviertes Wort, das nur in Elementfunktionen innerhalb von Klassen verwendet werden kann.

This initialisation is only partly bracketed
[Diese Initialisierung steht nur teilweise in geschweiften Klammern]
(Compiler, Warnung) Resultat der IDE-Auswahl *Optionen/Compiler/Meldungen.../Verstöße gegen ANSI*. Diese Initialisierung steht nur teilweise in geschweiften Klammern. Wenn Strukturen initialisiert werden, kann mit Hilfe der geschweiften Klammern jedes Element der Struktur gekennzeichnet werden. Wenn ein Element selbst ein Array oder eine Struktur ist, können verschachtelte geschweifte Klammern eingesetzt werden. Damit ist gewährleistet, daß der Compiler erkennt, welcher Wert zu welchem Element gehört. Wenn Sie dabei die geschweiften Klammern nicht vollständig angeben, erhalten Sie die vorliegende Meldung.

Too few arguments in template class name ***Schablone***
[Zu wenig Argumente in der Template-Klasse *Schablone*]
(Compiler, Fehler) In einem Template-Klassennamen fehlen Werte für formale Parameter.

Too few parameters in call
[Zu wenig Parameter im Aufruf]
(Compiler, Fehler) Der Aufruf einer Funktion mit einem Prototyp (über einen Funktionszeiger) hat zu wenig Argumente. Bei Prototypen müssen alle Parameter angegeben sein.

Too few parameters in call to ***Funktion***
[Aufruf von *Funktion* mit zu wenig Parametern]
(Compiler, Fehler) Eine zuvor mit einem Prototypen deklarierte bzw. bereits definierte Funktion wird hier mit zu wenig Argumenten aufgerufen.

Too many decimal points
[Zu viele Dezimalpunkte]
(Compiler, Fehler) Es ist eine Gleitkommakonstante mit mehr als einem Dezimalpunkt vorhanden.

Too many default cases
[Zu viele **default**-Anweisungen]
(Compiler, Fehler) In einer **switch**-Anweisung wurde mehr als eine **default**-Anweisung gefunden.

Too many error or warning messages
[Zu viele Fehlermeldungen oder Warnungen]
(Compiler, Fehler) Sie haben im Menü *Optionen/Compiler/Meldungen/Anzeige* zu viele Fehlermeldungen oder Warnungen festgelegt.

Too many error or warning messages
[Zu viele Fehlermeldungen und Warnungen]
(Linker, Fehler) Die Anzahl der Meldungen, die der Compiler ausgegeben kann, ist überschritten. Diese Meldung erhalten Sie dann, wenn der Linker seine Grenze erreicht hat.

Too many exponents
[Zu viele Exponenten]
(Compiler, Fehler) Es wurde mehr als ein Exponent in einer Gleitkommakonstante gefunden.

Too many initializers
[Zu viele Initialisierer]
(Compiler, Fehler) Es sind mehr Initialisierer vorhanden, als durch die Deklaration zugelassen sind.

Too many storage classes in declaration
[Zu viele Speicherklassen in der Deklaration]
(Compiler, Fehler) Eine Deklaration darf nicht mehr als eine Speicherklasse enthalten.

Too many types in declaration
[Zu viele Typen in der Deklaration]
(Compiler, Fehler) Eine Deklaration kann jeweils nur einen der Grundtypen **char**, **int**, **float**, **double**, **struct**, **union**, **enum** oder einen **typedef**-*Namen* enthalten.

Too much global data defined in file
[Zu viele globale Daten in der Datei]
(Compiler, Fehler) Der Gesamtumfang der globalen Datendeklarationen überschreitet 64 KByte. Überprüfen Sie die Deklarationen bei allen Arrays, die zu groß sein könnten. Ziehen Sie auch die Reorganisation des Programms oder die Verwendung von **far**-Variablen in Betracht, wenn alle Deklarationen notwendig sind.

Trying to derive a far class from the huge base *Basis*
[Es wurde versucht, eine Klasse vom Typ **far** von einer Basisklasse vom Typ **huge** abzuleiten]
(Compiler, Fehler) Wenn eine Klasse vom Typ **huge** deklariert (oder so voreingestellt) ist, müssen alle von ihr abgeleiteten Klassen ebenfalls vom Typ **huge** sein.

Trying to derive a far class from the near base *Basis*
[Es wurde versucht, eine Klasse vom Typ **far** von einer Basisklasse vom Typ **near** abzuleiten]
(Compiler, Fehler) Wenn eine Klasse vom Typ **near** deklariert (oder so voreingestellt) ist, müssen alle von ihr abgeleiteten Klassen ebenfalls vom Typ **near** sein.

Trying to derive a huge class from the far base *Basis*
[Es wurde versucht, eine Klasse vom Typ **huge** von einer Basisklasse vom Typ **far** abzuleiten]
(Compiler, Fehler) Wenn eine Klasse vom Typ **far** deklariert (oder so voreingestellt) ist, müssen alle von ihr abgeleiteten Klassen ebenfalls vom Typ **far** sein.

Trying to derive a huge class from the near base *Basis*
[Es wurde versucht, eine Klasse vom Typ **huge** von einer Basisklasse vom Typ **near** abzuleiten]
(Compiler, Fehler) Wenn eine Klasse vom Typ **near** deklariert (oder so voreingestellt) ist, müssen alle von ihr abgeleiteten Klassen ebenfalls vom Typ **near** sein.

Trying to derive a near class from the far base *Basis*
[Es wurde versucht, eine Klasse vom Typ **near** von einer Basisklasse vom Typ **far** abzuleiten]
(Compiler, Fehler) Wenn eine Klasse vom Typ **far** deklariert (oder so voreingestellt) ist, müssen alle von ihr abgeleiteten Klassen ebenfalls vom Typ **far** sein.

Trying to derive a near class from the huge base *Basis*
[Es wurde versucht, eine Klasse vom Typ **near** von einer Basisklasse vom Typ **huge** abzuleiten]
(Compiler, Fehler) Ist eine Klasse vom Typ **huge** deklariert (oder so voreingestellt), müssen alle von ihr abgeleiteten Klassen ebenfalls vom Typ **huge** sein.

Two consecutive dots
[Zwei aufeinanderfolgende Punkte]
(Compiler, Fehler) Eine Ellipse enthält drei Punkte (...), ein Dezimalpunkt oder ein Elementauswahl-Operator einen Punkt (.) – somit sind zwei aufeianderfolgende Punkte in einem C-Programm nicht erlaubt.

Two operands must evaluate to the same type
[Zwei Operanden müssen sich zum selben Datentyp auswerten lassen]
(Compiler, Fehler) Im Bedingungsoperator `?:` müssen die Typen der Ausdrücke auf beiden Seiten des Doppelpunkts dieselben sein, außer bei den gängigen Konvertierungen wie **char** in **int** oder **float** in **double** oder **void*** in einen Zeiger. Hier ergeben die beiden Seiten verschiedene Datentypen, die nicht automatisch konvertiert werden. Unter Umständen brauchen Sie bloß eine Seite in den Datentyp der anderen umzuwandeln.

Hinweis zur Familie der Type mismatch-Meldungen: Wenn Sie ein C++ Programm compilieren, geht den folgenden `Type mismatch`-Meldungen immer eine andere Meldung voraus (meist `Cannot convert Typ1 to Typ2`). Die im folgenden aufgeführten Meldungen geben danach den genauen Grund für die Art des Typ-Konflikts an.

Type mismatch in default argument value
[Typ-Konflikt im voreingestellten Argumentwert]
(Compiler, Fehler) Der voreingestellte Wert für den Parameter konnte nicht in

den Typ des Parameters konvertiert werden. Diese Meldung erhalten Sie, wenn der Parameter keinen Namen hat.

Type mismatch in default value for parameter ***Parameter***
[Typ-Konflikt im voreingestellten Wert für den Parameter *Parameter*]
(Compiler, Fehler) Der voreingestellte Wert für den Parameter konnte nicht in den Typ des Parameters konvertiert werden.

Type mismatch in parameter ***Nummer***
[Typkonflikt in Parameter *Nummer*]
(Compiler, Fehler) Die mit einem Funktionszeiger aufgerufene Funktion wurde mit einem Prototyp deklariert; der Parameter *Nummer* (gezählt wird von links nach rechts, ausgehend von 1) konnte nicht in den deklarierten Parametertyp konvertiert werden. (Vergleiche den obigen Hinweis.)

Type mismatch in parameter ***Nummer*** **in call to** ***Funktion***
[Typkonflikt in Parameter *Nummer* beim Aufruf von *Funktion*]
(Compiler, Fehler) Die angegebene Funktion wurde im Quelltext mit einem Prototyp deklariert; der Parameter *Nummer* (gezählt wird von links nach rechts, ausgehend von 1) konnte nicht in den deklarierten Parametertyp konvertiert werden. (Vergleiche den obigen Hinweis.)

Type mismatch in parameter ***Parameter***
[Typkonflikt in Parameter *Parameter*]
(Compiler, Fehler) Die mit einem Funktionszeiger aufgerufene Funktion wurde mit einem Prototyp deklariert; der angegebene Parameter konnte nicht in den deklarierten Parametertyp konvertiert werden. (Vergleiche den obigen Hinweis.)

Type mismatch in parameter ***Parameter*** **in call to** ***Funktion***
[Typkonflikt in Parameter *Parameter* beim Aufruf von *Funktion*]
(Compiler, Fehler) Die angegebene Funktion wurde im Quelltext mit einem Prototyp deklariert; der angegebene Parameter (gezählt wird von links nach rechts, ausgehend von 1) konnte nicht in den deklarierten Parametertyp konvertiert werden. Vergleiche die Erläuterung unter » *Type mismatch in parameter Parameter*«.

Type mismatch in parameter ***Parameter*** **in template class name** ***Schablone***
[Typkonflikt in Parameter *Parameter* im Template-Klassennamen Template]
(Compiler, Fehler) Der für den Parameter angegebene Wert des Template-Arguments stimmt nicht mit dem Typ des formalen Template-Parameters überein. (Vergleiche den obigen Hinweis.)

Type mismatch in parameter ***Nummer*** **in template class name** ***Schablone***
[Typkonflikt im Parameter *Nummer* im Template-Klassennamen Template]
(Compiler, Fehler) Der für den Parameter angegebene Wert des Template-Argu-

ments stimmt nicht mit dem Typ des formalen Template-Parameters überein. (Vergleiche den obigen Hinweis.)

Type mismatch in redeclaration of ***Bezeichner***
[Neudeklaration von *Bezeichner* mit anderem Typ]
(Compiler, Fehler) In Ihrem Quelltext wird eine Variable mit einem Typ erneut deklariert, der sich von dem ursprünglichen Variablentyp unterscheidet. Dies kann vorkommen, wenn eine Funktion aufgerufen und danach deklariert wird, um etwas anderes als einen Integerwert zurückzuliefern. Wenn dies der Fall ist, müssen Sie die Funktion vor ihrem ersten Aufruf deklarieren.

Type name expected
[Name des Datentyps erwartet]
(Compiler, Fehler) Einer der folgenden Fehler trat auf:

- Bei der Deklaration einer Variablen auf Dateiebene oder eines **struct**-Feldes wurde weder Datentyp noch Speicherklasse angegeben.
- Bei der Deklaration eines **typedef** wurde kein Datentyp angegeben.
- Bei der Deklaration eines Destruktors für eine C++ Klasse war der Destruktorname kein Typname – er muß dem Namen seiner Klasse entsprechen.
- Beim Angeben des Namens einer C++ Basisklasse war der Name nicht der einer Klasse.

Type qualifier ***Bezeichner*** **must be a struct or a class name**
[Der Typ-Qualifizierer *Bezeichner* muß ein **struct** oder ein Klassenname sein]
(Compiler, Fehler) Der C++ Qualifizierer in der Konstruktion *Qualifizierer::-Bezeichner* ist nicht der Name eines **struct** oder einer **class**.

Unable to create output file ***Dateiname***
[Ausgabedatei *Dateiname* kann nicht erzeugt werden]
(Compiler, schwerer Fehler) Die Diskette ist entweder voll oder schreibgeschützt oder das Ausgabeverzeichnis existiert nicht. Wenn die Diskette voll ist, löschen Sie nicht mehr benötigte Dateien und starten Sie die Compilierung erneut. Ist die Diskette schreibgeschützt, kopieren Sie die Quelltextdateien auf eine beschreibbare Diskette und starten Sie die Compilierung erneut.

Unable to open file ***Dateiname***
Unable to open ***Dateiname***
[Die Datei *Dateiname* kann nicht geöffnet werden]
(Linker, schwerer Fehler und Bibliotheksverwaltung, Fehler) Entweder existiert die angegebene Datei nicht oder ihr Name wurde falsch geschrieben.

Unable to open ***Dateiname*** **for output**
[Die Datei *Dateiname* kann nicht für die Ausgabe geöffnet werden]
(Bibliotheksverwaltung, Fehler) Die Bibliotheksverwaltung kann die angegebene Datei nicht für die Ausgabe öffnen. In diesem Fall fehlt entweder Speicher-

platz für die Zielbibliothek oder eine Listendatei. Dieser Fehler tritt auch dann auf, wenn die Zieldatei existiert, aber schreibgeschützt ist.

Unable to open include file ***Dateiname***
[#*include*-Datei *Dateiname* kann nicht geöffnet werden]
(Compiler, Fehler) Der Compiler kann die angegebene Datei nicht finden. Das kann daran liegen, daß eine Include-Datei eine #*include*-Anweisung für sich selbst enthält. Oder Sie haben in der Datei CONFIG.SYS in Ihrem Stammverzeichnis den Parameter FILES nicht gesetzt (versuchen Sie es mit `FILES=20`). Überprüfen Sie, ob die angegebene Datei existiert.

Unable to open input file ***Dateiname***
[Eingabedatei *Dateiname* nicht gefunden]
(Compiler, Fehler) Die Quelltextdatei kann nicht gefunden werden. Überprüfen Sie die Schreibweise des Namens und ob sich die Datei auf der richtigen Diskette bzw. im richtigen Verzeichnis befindet.

Unable to rename ***Dateiname*** **to** ***Dateiname***
[*Dateiname* kann nicht in *Dateiname* umbenannt werden]
(Bibliotheksverwaltung, Fehler) Die Bibliotheksverwaltung baut eine Bibliothek in einer temporären Datei auf und benennt diese Datei dann in den Dateinamen der Zielbibliothek um. Wenn dabei ein Fehler auftritt, liegt dies normalerweise an mangelndem Speicherplatz auf der Diskette/Platte.

Undefined label ***Bezeichner***
[Undefiniertes Label *Bezeichner*]
(Compiler, Fehler) Ein **goto** einer Funktion bezieht sich auf ein undefiniertes Label.

Undefined structure ***Bezeichner***
[Struktur *Bezeichner* nicht definiert]
(Compiler, Warnung) Die angegebene Struktur wurde in der Quelltextdatei verwendet, eventuell in einem Zeiger auf eine Struktur. Es gibt aber keine Definition für diese Struktur im Quelltext. Das könnte daran liegen, daß ein Strukturname falsch geschrieben wurde oder daß eine Deklaration fehlt.

Undefined structure ***Struktur***
[Struktur *Struktur* nicht definiert]
(Compiler, Fehler) In der Quelltextdatei wird die angegebene Struktur verwendet, aber es ist keine Definition für sie vorhanden. Der Fehler ist vermutlich einige Zeilen vor der ausgewiesenen Zeile aufgetreten (eventuell in einem Zeiger auf eine Struktur). Entweder enthält der Strukturname einen Tippfehler oder es fehlt eine Deklaration.

Undefined symbol ***Bezeichner***
[Symbol *Bezeichner* ist nicht definiert]
(Compiler, Fehler) Der angegebene Bezeichner wurde nicht deklariert. Es handelt sich dabei um einen Tippfehler an dieser Stelle oder in der Deklaration. Es

könnte aber auch daran liegen, daß ein Fehler in der Deklaration des Bezeichners vorhanden ist.

Undefined symbol *Symbol* **in module** *Modul*
[Undefiniertes Symbol *Symbol* im Modul *Modul*]
(Linker) Das angegebene Symbol ist im angegebenen Modul referenziert, aber nicht in den gelinkten .OBJ-Dateien und Bibliotheken definiert. Überprüfen Sie, ob das Symbol fehlerhaft geschrieben ist.

Bei Turbo C++ tritt dieser Linker-Fehler normalerweise dann auf, wenn die Symboldefinitionen vom Typ **pascal** und **cdecl** in unterschiedlichen Dateien nicht übereinstimmen oder wenn Sie einen erforderlichen Namen für eine .OBJ-Datei nicht angegeben haben. Wenn Sie C++ Module mit C-Modulen linken, haben Sie eventuell vergessen, externe C-Deklarationen in *extern "C" {...}* umzuschreiben.

Unexpected }
[Unerwartete }]
(Compiler, Fehler) Hier trat eine überflüssige rechte geschweifte Klammer auf. Prüfen Sie, wo die linke fehlt.

Unexpected end of file in comment started on *Zeilennummer*
[Unerwartetes Dateiende im Kommentar, der mit Zeile *Zeilennummer* beginnt]
(Compiler, Fehler) Die Quelltextdatei endet mitten in einem Kommentars. Hier fehlt normalerweise irgendwo der Abschluß des Kommentars mit */.

Unexpected end of file in conditional started on line *Zeilennummer*
[#*if*-Block vor Dateiende nicht abgeschlossen. Anfang der Bedingung in Zeile *Zeilennummer*]
(Compiler, Fehler) Das Ende einer Quelltextdatei wurde erreicht, bevor das zu einem #*if* gehörige #*endif* gefunden wurde. Das #*endif* fehlt oder wurde falsch geschrieben.

union cannot be a base type
[Eine Variante kann keine Basisklasse sein]
(Compiler, Fehler) Im allgemeinen darf eine C++ Klasse vom Typ **union** sein, sie kann aber nicht als Basis für andere Klassen verwendet werden.

union cannot have a base type
[Eine Variante kann keine Basisklasse haben]
(Compiler, Fehler) Eine Variante kann nicht von anderen Klassen abgeleitet werden.

Union member *Element* **is of type class with constructor**
[Element *Element* einer Variante ist vom Typ Klasse mit Konstruktor]
(Compiler, Fehler) Eine Variante darf keine Elemente enthalten, die vom Typ **class** sind und benutzerdefinierte Konstuktoren besitzen .

Union member *Element* **is of type class with destructor**
[Element *Element* einer Variante ist vom Typ Klasse mit Destruktor]
(Compiler, Fehler) Eine Variante darf keine Elemente enthalten, die vom Typ **class** sind und benutzerdefinierte Destruktoren besitzen .

Union member *Element* **is of type class with operator=**
[Element *Element* einer Variante ist vom Typ Klasse mit Operator =]
(Compiler, Fehler) Eine Variante darf keine Elemente enthalten, die vom Typ **class** sind und einen operator=besitzen .

Unions cannot have virtual member functions
[Varianten dürfen keine virtuellen Elementfunktionen enthalten]
(Compiler, Fehler) Eine Variante darf keine virtuellen Funktionen als Elemente enthalten.

Unknown assembler instruction
[Unbekannter Opcode in einer **asm**-Anweisung]
(Compiler, Warnung) Der Compiler hat eine Inline-Assembler-Anweisung gefunden, die einen unzulässigen Opcode enthält. Überprüfen Sie die Schreibweise des Opcodes (vgl. dazu Kapitel 7 »BASM und Inline-Assembler«). Diese Warnung ist standardmäßig ausgeschaltet.

Unkown command line switch *X* **ignored**
[Die unbekannte Kommandozeilenoption *X* wurde ignoriert]
(Bibliotheksverwaltung, Warnung) In der Kommandozeile oder in der Response-Datei wurde ein Schrägstrich (/) gefunden, auf den keine gültige Option folgt.

Unknown language, must be C or C++
[Sprache unbekannt, C oder C++erforderlich]
(Compiler, Fehler) In der C++Konstruktion

```
extern "name" type func( /*...*/ );
```

muß der in Anführungszeichen stehende Namen der Sprache C oder C++ entstammen, andere Sprachen werden nicht akzeptiert. Eine externe Pascalfunktion kann wie folgt deklariert werden, ohne daß der Compiler sie umbenennt:

```
extern "C" int pascal func( /*...*/ );
```

Eine (möglicherweise überladene) C++ Funktion kann als **pascal** deklariert werden, und das Umbenennen durch den Compiler zulassen:

```
extern int pascal func( /*...*/ );
```

Unreachable code
[Programmteil kann nie ausgeführt werden]
(Compiler, Warnung) Nach einer **break**-, **continue**-, **goto**- oder **return**-Anweisung folgt kein Label, kein Ende einer Schleife oder Funktion. Der Compiler überprüft **while**-, **do**- und **for**-Schleifen mit einer konstanten Testbedingung und versucht Schleifen festzustellen, die nie durchlaufen werden.

Unterminated string or character constant
[Zeichen/Stringkonstante nicht korrekt abgeschlossen]
(Compiler, Fehler) Nach einer String- oder Zeichenkonstante fehlt das abschließende Anführungszeichen.

Use . or ->to call ***Funktion***
[Aufruf einer Funktion mit . oder ->]
(Compiler, Fehler) Sie haben versucht, eine Elementfunktion aufzurufen, ohne ein Objekt anzugeben.

Use . or ->to call ***Element*** **or & to take its adress**
[Aufruf einer Elementfunktion mit **.** oder **->** Abruf ihrer Adresse mit **&**]
(Compiler, Fehler) Es wurde eine Referenz auf ein nichtstatisches Element einer Klasse gefunden. Ein solches Element kann aber nur durch Angabe eines Objekts oder Abruf seiner Adresse mit dem **&**-Operator verwendet werden.

Use :: to take the address of a member function
[Adresse einer Elementfunktion nur über den Operator **::**]
(Compiler, Fehler) Ist *f* eine Elementfunktion der Klasse *c*, erhält man deren Adresse über *&c::f*. Beachten Sie, daß ein Klassenname und nicht der eines Objekts verwendet wird, und daß ein doppelter Doppelpunkt zwischen Klasse und Funktion stehen muß. (Zeiger auf Elementfunktionen sind keine echten Zeigertypen und beziehen sich auf keine bestimmte Instanz einer Klasse).

Use qualified name to access nested type ***Typ***
[Für den Zugriff auf den geschachtelten Typ *Typ* muß ein qualifizierter Name verwendet werden]
(Compiler, Warnung) In älteren C++ Versionen sind in Klassen deklarierte **typedef**- und »Tag«-Namen im globalen Gültigkeitsbereich direkt sichtbar. In der neuesten C++ Version müssen diese Namen mit dem Qualifizierer *class::* versehen werden, wenn sie außerhalb des Gültigkeitsbereichs ihrer Klasse verwendet werden sollen. Damit älterer Programmtext compiliert werden kann, in dem ein derartiger Name eindeutig in einer einzigen Klasse definiert ist, kann *class::* weggelasssen werden. Turbo C++gibt dann die vorliegende Warnung aus.

User break
[Abbruch durch den Benutzer]
(Linker oder Compiler, Fehler) Sie haben während des Compilierens in der IDE *Abbruch* gedrückt. Diese Meldung dient nur der Bestätigung.

Value of type void is not allowed
[Wert vom Typ **void** ist unzulässig]
(Compiler, Fehler) Ein Wert vom Typ **void** ist eigentlich kein Wert und darf deshalb nicht verwendet werden, wo ein Wert erforderlich ist. Solche Stellen sind die rechte Seite einer Zuweisung, das Argument einer Funktion und die Steueranweisungen der Ausdrücke **if**, **for** und **while**.

Variable *Bezeichner* **is initialized more than once**
[Die Variable *Bezeichner* wird mehr als einmal initialisiert]
(Compiler, Fehler) Diese Variable wurde mehrmals initialisiert. Es ist zulässig, eine Variable auf Dateiebene öfter als einmal zu deklarieren, aber eine Initialisierung darf nur einmal erfolgen (auch wenn es sich um zwei gleiche Initialisierungen handelt).

'virtual' can only be used with member functions
[Nur Elementfunktionen dürfen als **virtual** deklariert sein]
(Compiler, Fehler) Ein Datenelement wurde mit dem Spezifizierer **virtual** deklariert – es dürfen aber nur Elementfunktionen so deklariert werden.

Virtual function *Funktion1* **conflicts with base class** *Basis*
[Virtuelle *Funktion1* kollidiert mit Basisklasse *Basis*]
(Compiler, Fehler) Eine virtuelle Funktion hat dieselben Argumenttyp wie die Basisklasse, liefert aber einen verschiedenen Ergebnistyp. Dies ist unzulässig.

virtual specified more than once
[Das Wort **virtual** tritt mehr als einmal auf]
(Compiler, Fehler) In C++ darf das reservierte Wort **virtual** in der Deklaration einer Elementfunktion nur einmal auftreten.

void & is not a valid type
[**void &** ist kein gültiger Datentyp]
(Compiler, Fehler) Eine Referenz bezieht sich immer auf ein Objekt, ein Objekt kann aber nicht vom Typ **void** sein. Aus diesem Grund ist **void** nicht erlaubt.

Void functions may not return a value
[**void**-Funktion darf keinen Wert zurückliefern]
(Compiler, Warnung) Ihr Quelltext deklariert den Rückgabewert der aktuellen Funktion als **void**, der Compiler findet jedoch eine **return**-Anweisung mit Wert. Dieser Wert wird ignoriert.

Funktion **was previously declared with the language** *Sprache*
[*Funktion* wurde ursprünglich mit der Sprache *Sprache* deklariert]
(Compiler, Fehler) Es darf nur eine Sprache für eine **extern**-Funktion verwendet werden. Diese Funktion wurde an verschiedenen Positionen in demselben Modul mit unterschiedlichen Sprachen deklariert.

While statement missing (
[In der **while**-Schleife fehlt ( ]
(Compiler, Fehler) Der Compiler findet in einer **while**-Anweisung nach dem Schlüsselwort **while** keine linke Klammer.

While statement missing)
[In der **while**-Schleife fehlt)]
(Compiler, Fehler) Der Compiler findet in einer **while**-Anweisung nach dem Testausdruck keine rechte Klammer.

Wrong number of arguments in call of macro *Makro*
[Falsche Anzahl von Argumenten im Makro-Aufruf]
(Compiler, Fehler) Der Quelltext hat den Namen eines Makros mit einer falschen Anzahl von Argumenten aufgerufen.

A. HC: Der Hilfe-Compiler von Windows

Ein Hilfesystem stellt dem Anwender Online-Informationen über eine Anwendung zur Verfügung. Das Anlegen des Hilfesystems gliedert sich in zwei Aufgabenbereiche:

Entwicklung: Planung, Erfassung, Codierung, Aufbau und Überwachung der Themendateien. In diesen Dateien (es handelt sich um Textdateien) sind die verschiedenen Aspekte der Anwendung beschrieben.

Programmierung: Es muß sichergestellt werden, daß das Hilfesystem mit der Anwendung korrekt zusammenarbeitet.

Das vorliegende Kapitel befaßt sich mit folgenden Themen:

- Planen des Hilfesystems;
- Erstellen der Themendateien;
- Aufbau der Hilfedatei;
- Hilfe-Beispiele und Fehlermeldungen des Compilers.

Wir gehen bei diesem und allen folgenden Kapiteln davon aus, daß Sie mit dem Hilfesystem von Microsoft Windows vertraut sind. Sie finden in den einzelnen Kapiteln Anwendungsbeispiele, die auf Ihren Originaldisketten mitgeliefert werden. Sollten Sie das Windows-Hilfesystem nicht kennen, dann compilieren und starten Sie die Beispielanwendung Helpex.

Das Hilfesystem anlegen: Der Entwicklungszyklus

Das Anlegen eines Hilfesystems für eine Windows-Anwendung umfaßt folgende Hauptaufgaben:

1. Sammeln von Informationen für die Hilfethemen.

2. Planen des Hilfesystems. Der Abschnitt »Das Hilfesystem planen« befaßt sich mit den Aspekten, die Sie bei der Planung des Systems berücksichtigen müssen.

3. Erfassen der Hilfethemen.

4. Eingeben der erforderlichen Steuercodes in die Textdateien. Steuercodes legen fest, wie sich der Anwender im Hilfesystem »bewegen« kann. Der Abschnitt »Das Hilfesystem und der Autor«, beinhaltet ein Beispiel mit verschiedenen Steuercodes. Im Abschnitt »Themendateien erstellen« werden die Steuercodes im Detail beschrieben.

5. Anlegen einer Projektdatei für den Aufbau des Hilfesystems. Die Hilfe-Projektdatei enthält Informationen, die der Hilfe-Compiler für den Aufbau einer Hilfe-Ressourcendatei benötigt. Im Abschnitt »Die Hilfedatei compilieren« finden Sie Erläuterungen zu diesem Punkt.

6. Aufbau der Hilfe-Ressourcendatei. Die Hilfe-Ressourcendatei ist eine compilierte Version der Themendateien, die der Autor erfaßt hat. Im Abschnitt »Die Hilfedatei compilieren« wird die Compilierung einer Hilfe-Ressourcendatei erläutert.

7. Testen und debuggen des Hilfesystems.

8. Programmierung der Anwendung, so daß diese auf das Windows-Hilfesystem zugreifen kann.

Das Hilfesystem und der Anwender

Für den Anwender stellt sich das Hilfesystem als Teil der Anwendung dar. Es besteht aus Text und Grafiken, die in Form eines Hilfefensters über die Anwendung gelegt werden.

Abbildung A.1 zeigt das Hilfefenster, das geöffnet wird, wenn der Anwender im Beispielprogramm Helpex nach Hilfe über das Kopieren von Text verlangt hat.

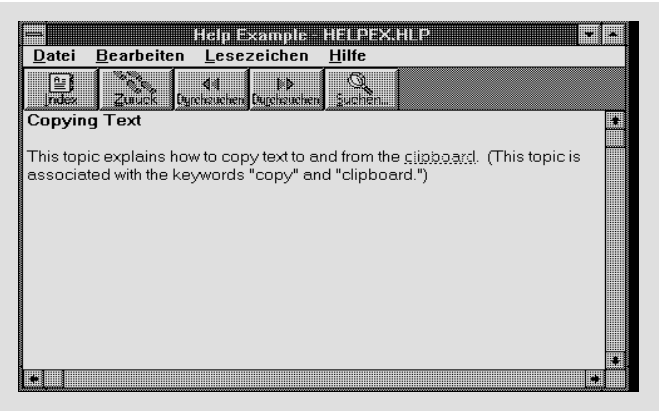


Abbildung A.1
Das Hilfefenster

Das Hilfefenster enthält ein Beispiel für ein Hilfethema, das eine auszuführende Aufgabe teilweise beschreibt. Der erste Satz liefert die Definition des Wortes »Zwischenablage«. Steht der Zeiger auf dem unterstrichenen Wort und

drückt der Anwender die Maustaste, wird ein zusätzliches Fenster (Detailfenster) mit Informationen angezeigt. Dieses zweite Fenster bleibt solange geöffnet, bis der Anwender die Maustaste wieder losläßt.

Kreuzreferenzen zu verwandten Themen werden *Sprünge* genannt. Durch Anklicken eines Sprungbegriffs (markiert durch die gepunktete Unterstreichung) wird im Hilfefenster die Beschreibung dieses verwandten Themas oder Befehls dargestellt.

Das Hilfesystem und der Autor

Für den Autor stellt sich das Hilfesystem als eine Gruppe von Themendateien dar. Diese Dateien enthalten sowohl Text als auch spezielle Steuercodes.

Abbildung A.2 zeigt den Quelltext, der dem Thema in Abbildung A.1 entspricht.

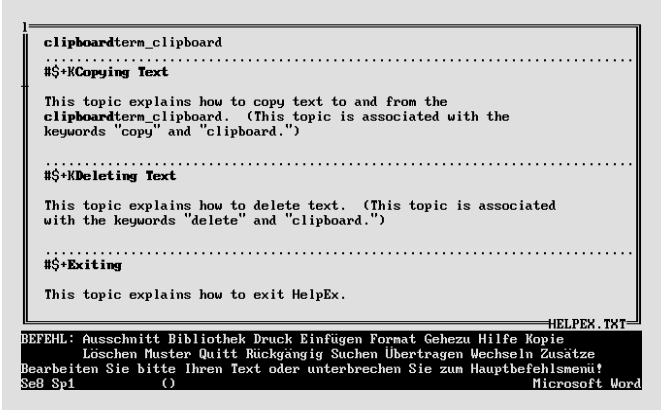


Abbildung A.2 Themendatei

Der Autor beschreibt für das Hilfethema die Aufgabe, formatiert den Text und fügt Steuercodes für durchgestrichenen und unterstrichenen Text sowie für Fußnoten ein. Unterstützt das verwendete Textprogramm keinen durchgestrichenen Text, kann stattdessen auch eine doppelte Unterstreichung verwendet werden. Fußnoten enthalten verknüpfende Information, die für den Hilfe-Compiler erforderlich ist. Im Abschnitt »Das Hilfesystem planen« erfahren Sie Näheres zu der Formatierung. Der Abschnitt »Themendateien erstellen« befaßt sich mit der Erstellung von Themen und der Eingabe von speziellen Codes für das Hilfesystem.

Das Hilfesystem und der Programmierer

Für den Programmierer ist das Hilfesystem eine selbständige Windows-Anwendung, die der Anwender wie jede andere Anwendung benutzen kann.

Ihre Anwendung kann die *WinHelp*-Funktion aufrufen. Diese Funktion veranlaßt Windows, das Hilfesystem zu starten und bestimmt, welches Thema im Hilfefenster angezeigt werden soll.

Im Abschnitt »Die Hilfedatei compilieren« finden Sie Einzelheiten über die Schnittstelle (API) zum Hilfesystem.

Das Hilfesystem planen

Der erste Arbeitsschritt bei der Entwicklung eines Hilfesystems ist die Ausarbeitung eines Rahmenplans. In diesem Abschnitt diskutieren wir das Planen eines Hilfesystems für eine spezielle Anwendung.

Der Abschnitt befaßt sich mit:

- der Entwicklung des Plans;
- der Bestimmung der Themendatei-Struktur;
- dem Entwurf für die Darstellung der Hilfethemen auf dem Bildschirm.

Einen Plan entwickeln

Bevor Sie Hilfethemen – ausgehend von den gesammelten Informationen – erfassen, sollten Sie mit den anderen an der Erstellung des Systems beteiligten Personen einen Plan erarbeiten, der folgende Aspekte berücksichtigt:

- Anwenderkreis Ihrer Anwendung;
- Inhalt der Hilfethemen;
- Strukturierung der Themen;
- Einsatz von kontextabhängigen Themen.

Sie sollten Ihren Plan aufzeichnen und darin einen Abriß der Hilfeinformation, ein Diagramm der Themenstruktur und Beispiele für die verschiedenen Themenarten aufführen.

Bedenken Sie, daß ein kontextabhängiges Hilfesystem erhöhte Entwicklungszeiten verursacht, vor allem für den Anwendungsprogrammierer.

Anwenderkreis definieren

Der Kreis von Anwendern, an den sich Ihre Anwendung richtet, bestimmt die Art der im Hilfesystem zur Verfügung stehenden Informationen und die Form der Darstellung dieser Informationen.

Anwender von Hilfesystemen lassen sich in folgende Kategorien einteilen:

Tabelle A.1 Anwenderkreis

Anwender	Vorwissen
Absolute EDV-Anfänger	Jemand, der noch keinerlei EDV-Kenntnisse besitzt.
Anwendungs-Anfänger	Es sind EDV-Kenntnisse beim Anwender vorhanden, jedoch nicht für Ihre Art der Anwendung. So kann z.B. jemand mit Textprogrammen vertraut sein, aber keinerlei Wissen über Kalkulationsprogramme mitbringen.
Anwendungs-Kenner	Der Anwender hat bereits mit Ihrer Anwendungsart gearbeitet und kann damit umgehen.
Anwendungs-Experte	Der Anwender besitzt gründliche Kenntnisse über Ihre Anwendungsart.

Inhalt des Hilfesystems planen

Sie sollten genügend und entsprechend spezielle Themen erfassen, damit der Anwender auch die Informationen erhält, die er benötigt.

Absolute EDV-Anfänger benötigen Hilfestellungen bei dem Erlernen von Operationen und dem Vertrautwerden mit Begriffsdefinitionen. Fortgeschrittenere Anwender werden nur ab und zu Hilfe bei einer Prozedur oder einem Begriff benötigen. In dem meisten Fällen ist das Hilfesystem für diesen Anwenderkreis eine Gedächtnisstütze für Befehle und Funktionen. Experten werden sich nur über die Syntax von Befehlen und Funktionen, Tastaturentsprechungen und Tastenkürzel informieren.

Für den Inhalt eines Hilfesystem gibt es keine festen Regeln. Wenn Ihr Hilfesystem für Anwender mit unterschiedlichsten Vorkenntnissen geeignet sein soll, müssen Sie Befehle, Prozeduren, Definitionen, Eigenschaften, Funktionen – also alle für Ihre Anwendung relevanten Aspekte – beschreiben. Der angesprochene Anwenderkreis entscheidet letztlich darüber, welche Themen Sie in das Hilfesystem aufnehmen.

Wenn Sie sich für ein kontextabhängiges Hilfesystem entscheiden, müssen Sie sich über den Mehraufwand an Zeit im Klaren sein. Es muß eine enge Zusammenarbeit zwischen dem Autor der Hilfestellung und dem Anwendungsprogrammierer gewährleistet sein, die mehr Entwicklungszeit zur Folge hat.

Struktur der Hilfethemen planen

In den meisten Hilfesystemen sind die Themen hierarchisch angeordnet. An der Spitze der Hierarchie steht ein Index und/oder ein Inhaltsverzeichnis. Die für den Anwender zur Verfügung stehenden Themen oder Themenkategorien werden im Index und Inhaltsverzeichnis einzeln aufgeführt.

Auch die Themen selbst können hierarchisch miteinander verknüpft sein. Jeder Abfrageschritt führt den Anwender eine Stufe tiefer in der Hierarchie, bis er schließlich die gewünschte Information erhält. Die hierarchische Verknüpfung der Hilfethemen bestimmt zum Teil die Art, in der sich der Anwender durch das Hilfesystem bewegen kann. Abbildung A.3 zeigt eine mögliche Hierarchie:

Helpex enthält einen Index, in dem die verschiedenen Themenkategorien

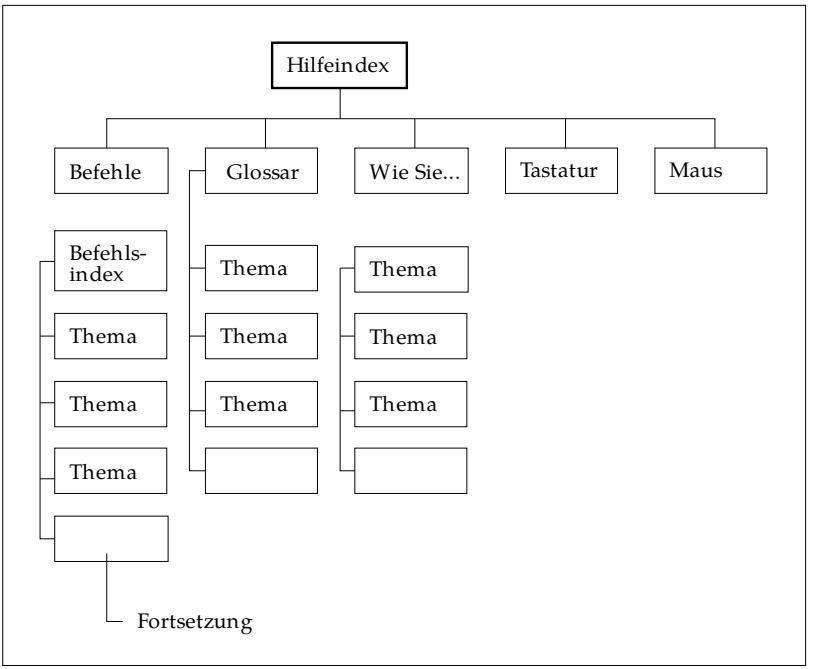


Abbildung A.3 Beispiel für eine Hilfe-Hierarchie

aufgeführt sind. Zu jeder Kategorie gehört ein Sekundärindex, der die Themen in der Kategorie und die Themen selbst auflistet.

Ein Sprung vom Index zum Thema bedeutet für den Anwender einen Sprung vom Generellen zum Speziellen.

Die hierarchische Struktur stellt dem Anwender einen Referenzpunkt zur Verfügung. Der Anwender muß in der Hierarchie nicht von oben nach unten vorgehen; er kann quer durch die Hierarchie von einem Thema zum anderen springen. Die hierarchische Verknüpfung wird durch die Sprungmöglichkeiten sozusagen verdeckt. Beispielsweise enthält das Windows-Hilfesystem eine Suchfunktion, in der der Anwender ein Schlüsselwort in ein Dialogfenster eingeben und damit nach verwandten Themen suchen kann. Das Hilfesystem zeigt dann eine Liste mit auswählbaren Titeln an, mit deren Hilfe auf die mit dem Schlüsselwort verwandten Informationen zugegriffen werden kann.

Da der Anwender in sehr vielen Fällen genau weiß, zu welchem Punkt er Informationen benötigt, kann er das gewünschte Thema schneller mit der Suchfunktion finden als über das Durcharbeiten der Hierarchie.

Weitere Informationen über die Suchfunktion finden Sie im Abschnitt »Schlüsselwörter zuweisen« weiter hinten in diesem Kapitel.

Wenn Sie die Themen nicht hierarchisch anordnen möchten, haben Sie die Möglichkeit, die Themen in einer dem Anwenderkreis angepaßten logischen Abfolge anzulegen. Bei einer logischen Abfolge, der sog. »Such-Sequenz«, kann der Anwender mit den »Durchsuchen«-Schaltern Themen durchsuchen. Such-Sequenzen sind vor allem für Anwender wichtig, die mehrere miteinander

verwandte Themen gleichzeitig lesen möchten, wie z.B. Themen, die sich mit den Befehlen des Datei-Menüs befassen. Weitere Informationen hierzu finden Sie im Abschnitt »Themendateien erstellen«.

Unabhängig von der ausgewählten Struktur sollten Sie die Anzahl der Listen, durch die sich der Anwender bis zu der gewünschten Information arbeiten muß, möglichst niedrig halten. Ebenso sollten Sie darauf achten, daß der Anwender nicht zu viele Hierarchie-Ebenen bis zum gewünschten Thema überwinden muß. Die meisten Hilfesysteme kommen sehr gut mit zwei oder drei Ebenen aus.

Kontextabhängige Hilfethemen anzeigen

Das Windows-Hilfesystem unterstützt die Erstellung kontextabhängiger Hilfesysteme. Wird das kontextabhängige Hilfesystem zusammen mit der Anwendungsprogrammierung geschrieben, kann der Anwender in einem geöffneten Menü die Taste F1 drücken, um sich für den aktuellen Menüeintrag Informationen anzeigen zu lassen. Alternativ dazu kann der Anwender Umschalt F1 drücken und dann einen Bildschirmbereich oder einen Befehl anklicken.

Die Entwicklung eines kontextabhängigen Hilfesystems muß in Zusammenarbeit von Hilfesystem-Autor und Anwendungsprogrammierer erfolgen, damit das Hilfesystem und die Anwendung die Informationen korrekt austauschen.

Bei der Planung eines derartigen Systems müssen sich Autor und Programmierer auf eine Liste mit Kontextnummern einigen. Kontextnummern sind Zahlen, die sich auf Menübefehle oder Bildschirmbereiche der Anwendung beziehen. Mit Hilfe dieser Zahlen werden die Verknüpfungen zwischen den markierten Befehlen und Bereichen und den Hilfethemen hergestellt. Sie können dann diese Nummern mit den entsprechenden Kontextstring-Bezeichnern in die Hilfe-Projektdatei eingeben, die der Hilfe-Compiler für den Aufbau der Hilfe-Ressourcendatei benutzt. Im Abschnitt »Hilfe-Projektdatei erstellen« finden Sie ausführliche Erläuterungen zu diesem Vorgang.

Die in der Hilfe-Projektdatei zugewiesenen Kontextnummern müssen denjenigen entsprechen, die die Anwendung zur Laufzeit beim Aufruf eines bestimmten Themas sendet. Details dazu finden Sie im Abschnitt »Schnittstelle zum Hilfesystem programmieren«. Weisen Sie den Hilfethemen keine expliziten Kontextnummern zu, erzeugt der Hilfe-Compiler Standardwerte, indem er die Kontextstrings in Kontextnummern konvertiert. Im Abschnitt »Kontextabhängige Themen anpassen: Der [Map]-Abschnitt« ist dieser Punkt näher erläutert.

Zur Verwaltung der Kontextnummern und Dateiinformationen können Sie sich ein Hilfeprotokoll aufbauen, das die Nummern der kontextabhängigen Hilfethemen enthält. Lesen Sie hierzu den Abschnitt »Hilfeprotokoll anlegen«.

Themendateistruktur festlegen

Die Struktur der Hilfedatei bleibt im Prinzip für alle Anwendungen gleich, auch wenn die Kontexte und Nummern der Themen unterschiedlich sind. Die einzelnen Themen in der Themendatei werden durch Seitenumbrüche voneinander getrennt. Beim Aufbau des Hilfesystems verwendet der Compiler diese Themendateien zur Erstellung der Information, die der Anwender im Hilfe-

fenster der Anwendung sieht. Abbildung A.4 illustriert die Grundstruktur einer Hilfedatei.

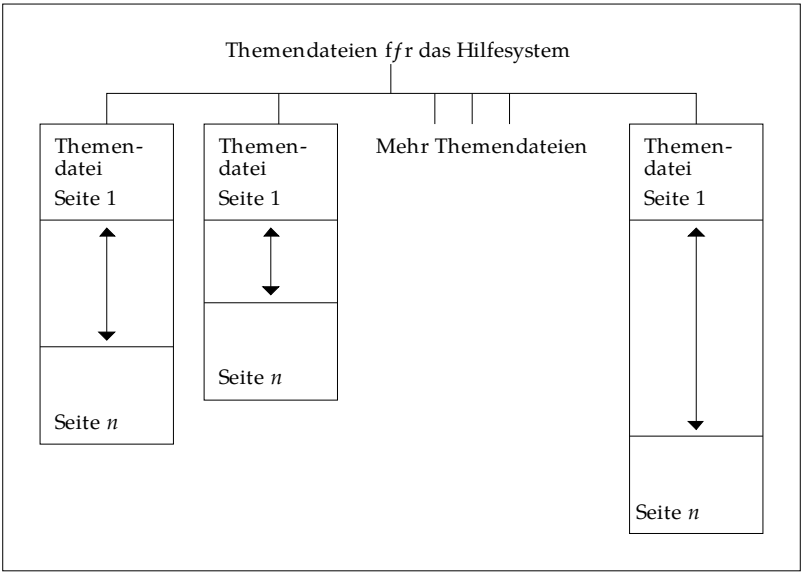


Abbildung A.4 Grundstruktur einer Hilfedatei

Eine Dateistruktur für Ihre Anwendung auswählen

Bei der Auswahl einer Dateistruktur für das Hilfesystem müssen Sie Umfang und Inhalt des Systems in Betracht ziehen. So könnten Sie z.B. alle Hilfethemen in eine einzige große Themendatei einbringen oder jedes Hilfethema in eine eigene Datei schreiben.

Aber: Keine der beiden Möglichkeiten ist vernünftig!

Eine zu große Einzeldatei oder zu viele separate Dateien verursachen bei der Erstellung der Hilfe-Ressourcendatei Probleme. Die Anzahl der Themen steht in direktem Zusammenhang mit der Anzahl der im Hilfesystem behandelten Merkmale. Folglich müssen bei einer umfangreichen Änderung der Themen auch die Merkmale angepaßt werden und umgekehrt. Wenn Sie zum Beispiel zusätzliche Produkt-Eigenschaften in das Hilfesystem aufnehmen, müssen Sie auch zusätzliche Themen erfassen, um die den neuen Informationen entsprechen.

Abbildung A.5 zeigt die Dateistruktur eines Hilfesystems. Die Anzahl der Themen und Themendateien wurde eingeschränkt, damit das Diagramm einfach und übersichtlich ist und die Verknüpfung der Themen durch Sprünge (dargestellt durch die Pfeillinien) deutlich wird. In der Abbildung soll nicht die Anzahl der Dateien dargestellt werden, die in ein Hilfesystem aufgenommen werden können. Die Abbildung betrifft ebenfalls nicht die Ordnung der Themendateien für den Einsatz von Such-Sequenzen.

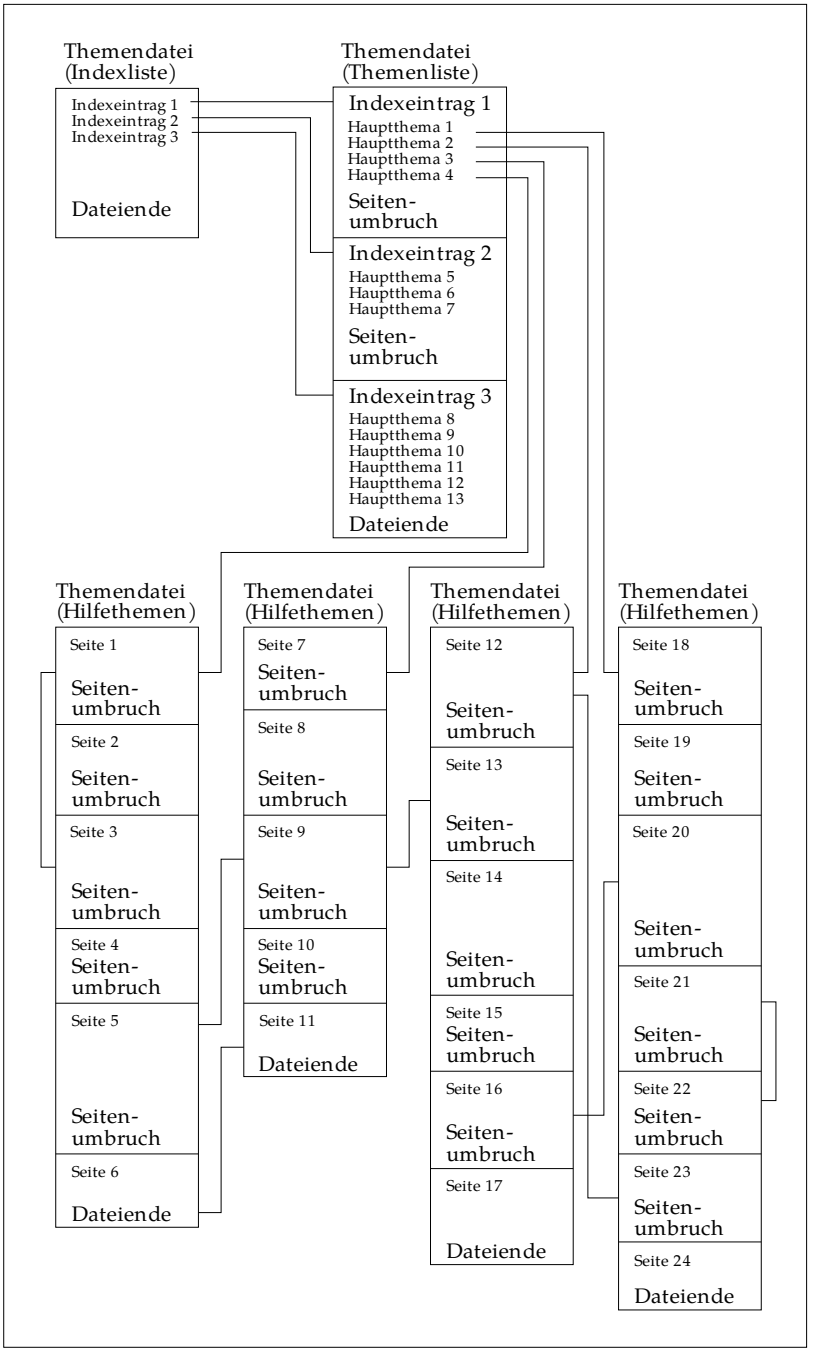


Abbildung A.5 Hilfedatei-Struktur mit Sprüngen

Hilfethemen auf dem Bildschirm darstellen

Die Form, in der der Anwender die Informationen auf dem Bildschirm sieht, hängt davon ab, welches Layout Sie für die Hilfethemen wählen. Das Hilfesystem unterstützt Textattribute und Grafiken, die Sie für die Gestaltung des Hilfefensters verwenden können. Im vorliegenden Abschnitt finden Sie generelle Richtlinien für den Entwurf eines Hilfefensters und eine Beschreibung der Schriften und Grafiken, die von dem Hilfesystem unterstützt werden.

Layout für den Hilfetext

Die Dateien mit den Hilfethemen müssen nicht nur aus unformatiertem Text bestehen. Sie können verschiedene Schriften und Schriftgrößen sowie Farben und Grafiken zur Hervorhebung wichtiger Punkte verwenden, Absätze einziehen, um komplexe Informationen abzuheben, und weitere optische Mittel zur Aufbereitung der Informationen einsetzen.

Untersuchungen, die sich mit Bildschirmformaten und Hilfesystemen beschäftigt haben, erbrachten generelle Richtlinien für die Präsentation von Anwenderinformationen. In der folgenden Tabelle haben wir die wichtigsten Ergebnisse der Untersuchungen für Sie zusammengestellt.

Tabelle A.2 Layout-Richtlinien

Layout	Richtlinie
Formulierung	*Berücksichtigen Sie bei der sprachlichen Darstellung immer den Anwenderkreis*, an den Sie sich wenden. Zu komplexe Formulierungen können Ihre Anwender frustrieren, wenn dadurch das Erlernen von Definitionen, unbekannten Begriffen und Konzepten erschwert wird.
Textumfang	*Halten Sie die Textmenge so gering wie möglich.* Untersuchungen haben ergeben, daß der Mensch für das Lesen von Bildschirminformationen ca. 30 Prozent mehr Zeit benötigt als für das Lesen von gedruckten Informationen. Knapper Text hilft, den erhöhten Zeitaufwand auszugleichen.
Absatzlänge	*Verwenden Sie kurze Absätze.* Textuelle Bildschirminformationen »erschlagen« den Leser schneller als gedruckter Text. Gliedern Sie deshalb den Text in kurze Absätze.
Abstände	*Gruppieren Sie Informationen, indem Sie Abstände freilassen.* Abstände machen Text leichter lesbar. Gehen Sie großzügig damit um, beachten Sie dabei aber immer die Relation zwischen Text und Bildschirmbgröße. Der Leser kann den Eindruck gewinnen, daß mehr Information auf dem Bildschirm steht, als in Wirklichkeit vorhanden ist. Wenn das Verhältnis von freiem Platz und Text 50:50 ist, nimmt der Leser ein Verhältnis von 40:60 wahr.
Hervorhebungen	*Gehen Sie mit Hervorhebungen gezielt um.* Das Hilfesystem stellt eine Vielzahl von Möglichkeiten zur Hervorhebung von Text zur Verfügung: verschiedene Schriften, Schriftgrößen und Farben.

Layout	Richtlinie
	Ein gezielter (sparsamer) Einsatz dieser Möglichkeiten beschleunigt das Auffinden von Informationen. Zu viele Hervorhebungen machen diesen Vorteil jedoch wieder zunichte und stören den Lesefluß. Verwenden Sie – ebenso wie bei Druckerzeugnissen – nur ein oder zwei Schriftarten gleichzeitig.
Grafiken und Symbole	*Setzen Sie Grafiken ein, wenn sich bestimmte Dinge damit leichter erklären lassen.* Das Windows-Hilfesystem unterstützt den Einsatz von Bitmap-Grafiken. Sie können damit beispielsweise die Funktion von Symbolen und Bildschirmelementen in Ihrer Anwendung erläutern. Denken Sie daran, daß der Leser zuerst die Bilder und dann den Text sieht. Gestalten Sie darum die Grafiken so, daß sie nicht mit Nebensächlichkeiten überladen sind. Verwenden Sie Farben nur dann, wenn das System des Anwenders Farben unterstützt. Denken Sie aber daran, daß die Erstellung von genauen und sinnvollen Grafiken – wie die Entwicklung von kontextabhängigen Hilfesystemen – mehr Zeit beansprucht.
Konsistenz	*Ihr Entwurf muß unter allen Umständen konsistent sein.* Der Anwender erwartet, daß alle Themen auf dieselbe Art und Weise präsentiert werden – unabhängig vom Inhalt der Informationen. Seien Sie darum bei der Gestaltung von Titeln, bei der Auswahl von Hervorhebungen und Schriften sowie bei der Positionierung von Text im Hilfefenster konsistent.

Schriften und Schriftgrößen

Das Windows-Hilfesystem kann Text in jeder Schrift und jeder Schriftgröße, die im System verfügbar ist, anzeigen. Beim Aufbau der Themendateien versucht der Hilfe-Compiler die vorgefundenen Schriften und Schriftgrößen einzusetzen. Kann eine Schrift oder Schriftgröße bei der Anzeige der Hilfedatei nicht verwendet werden, wird die Entsprechung des Anwendersystems herangezogen, die der gewählten Schrift oder Schriftgröße am nächsten kommt.

Windows wird nur mit einer begrenzten Anzahl von Schriften und Schriftgrößen ausgeliefert. Wenn Sie diese zur Erstellung der Hilfedateien verwenden, entspricht die angezeigte Hilfedatei genau der ausgedruckten Textprogrammdatei. Da andere als die mitgelieferten Schriften eventuell nicht auf dem Anwendersystem verfügbar sind, sollten Sie auch nur diese Windows-Schriften verwenden. Folgende Schriften gehören zum Lieferumfang von Windows:

Tabelle A.3 Windows-Schriften

Schrift	Größen	Schrift	Größen
Courier	10, 12, 15	Script	
Helvetica	8, 10, 12, 14, 18, 24	Symbol	8, 10, 12, 14, 18, 24
Modern		Tms Rmn	8, 10, 12, 14, 18, 24
Roman			

Da das Hilfesystem alle Windows-Schriften unterstützt, können auch spezielle Symbole, wie z.B. Pfeile, in den Hilfethemen enthalten sein. Verwenden Sie dazu die Schrift Symbol.

Grafikbilder

Das Hilfesystem ermöglicht es Ihnen, Grafiken in die Hilfedatei einzubinden. Grafiken können an einer beliebigen Position auf der Seite plaziert werden, und der Text kann danebenstehen.

Sie können auch farbige Grafiken einbringen, solange Sie die Windows-Farben verwenden. Wenn Sie Grafik-Werkzeuge einsetzen, die eine erweiterte Farbpalette zur Erstellung und Weitergabe von Grafiken unterstützen, werden die Grafiken unter Umständen nicht in den erwarteten Farben angezeigt. Da die Farbgegebenheiten auf einem Anwendersystem nicht von Ihnen kontrollierbar sind, sollten Sie sich bei Ihren Grafikbildern auf schwarzweiße Bitmaps beschränken.

Bedenken Sie, daß Grafiken immer dann sehr nützlich sind, wenn sie den Lernprozeß unterstützen. Grafiken, die nicht in engem Zusammenhang mit den Informationen stehen, wirken dagegen eher störend. Verzichten Sie auf derartige Bilder. Im Abschnitt »Themendateien erstellen« finden Sie unter »Grafikbilder einfügen« weitere Informationen über das Einbringen von Grafiken in die Hilfedatei.

Folgende Bücher enthalten Informationen zum Thema »Präsentation auf dem Bildschirm«:

Bradford, Annette Noris: Conceptual Differences Between the Display Screen and the Printed Page. In: Technical Communication (Third Quarter, 1984), S. 13 - 16.

Galitz, Wilbert O.: Handbook of Screen Format. Design. 3. Auflage, Wellesley, MA: QED Information Sciences, Inc., 1989.

Houghton, Raymond C., Jr.: Online Help Systems: A Conspectus. In: Communications of the ACM 27, Februar 1984, S. 126 - 133.

Queipo, Larry: User Expectations of Online Information. In: IEEE Transactions on Professional Communications PC 29, Dezember 1986, S. 11 - 15.

Themendateien erstellen

Der zeitintensivste Vorgang bei der Entwicklung eines Hilfesystems für Ihre Anwendung dürfte die Erstellung der Dateien mit den Hilfethemen sein, aus denen das Hilfesytem aufgebaut wird. Themendateien sind Textdateien, welche Beschreibungen enthalten, die dem Anwender nach dem Aufruf der Hilfefunktion angezeigt werden. In diesen Dateien werden verschiedene Informations-

arten definiert, wie beispielsweise Indizes, Befehlslisten oder Erläuterungen zur Ausführung von Aufgaben.

Zur Erstellung der Themendateien gehört die Erfassung des Textes, den der Anwender beim Aufruf der Hilfe-Funktion sieht, und die Eingabe der Steuercodes, die festlegen, wie der Anwender von Thema zu Thema gelangen kann. Im vorliegenden Abschnitt werden folgende Aspekte erläutert:

- Auswahl eines geeigneten Textprogramms;
- Strukturierung der Themendateien;
- Codierung der Themendateien;
- Verwaltung der Themendateien.

Geeignetes Textprogramm auswählen

Zur Erfassung der Textdateien benötigen Sie ein »Rich Text Format«-Textprogramm (RTF-Editor), der das Anlegen von Fußnoten, unterstrichenem, durchgestrichenem bzw. doppelt unterstrichenem Text als Kennzeichnung für die Steuercodes erlaubt.

Die Steuercodes werden im Abschnitt »Themendateien codieren« weiter hinten in diesem Kapitel erläutert. Aufgrund der RTF-Fähigkeit Word können Sie den codierten Text so eingeben, wie es die Definition der Hilfebegriffe (Sprünge, Schlüsselwörter) erfordert.

Themendateien strukturieren

Eine Themendatei enthält normalerweise mehrere Hilfethemen. Die einzelnen Themen müssen folgende Kriterien erfüllen:

- Themen werden durch »harte« Seitenumbrüche voneinander getrennt.
- Jedem Thema, auf das über eine Hypertext-Verbindung zugegriffen werden kann, muß ein eindeutiger Bezeichner oder Kontextstring zugewiesen werden.
- Themen können einen Titel haben.
- Themen können eine verknüpfte Liste von Schlüsselwörtern besitzen.
- Themen können einen symbolischen Namen (engl. *build tag*) haben.
- Zu jedem Thema kann eine logische Such-Sequenz gehören.

Wie Seitenumbrüche zwischen Themen eingefügt werden, können Sie in der Dokumentation Ihres Textprogramms nachlesen. Informationen über die Zuweisung von Kontextstrings und Titeln zu einzelnen Themen finden Sie in den nachfolgenden Abschnitten.

Themendateien codieren

In folgender Tabelle sind die Steuercodes des Hilfesystems zusammengestellt.

Tabelle A.4 Steuercodes des Hilfesystems

Steuercode	Funktion
Stern-Fußnote (*)	Symbolischer Name. Definiert einen Namen, mit dem Themen markiert werden, die der Compiler unter bestimmten Bedingungen in das System einbaut. Symbolische Namen sind optional. Sie müssen aber – wenn vorhanden – als erster Eintrag in einem Thema stehen.
Nummernzeichen (#)	Kontextstring. Definiert einen Kontextstring, der das Thema auf eindeutige Weise identifiziert. Da Hypertext auf Verknüpfungen basiert, die von Kontextstrings abhängig sind, können Themen ohne Kontextstrings nur über Schlüsselwörter oder die Such-Sequenz erreicht werden.
Dollarzeichen-Fußnote ($)	Titel. Definiert den Titel eines Themas. Titel sind optional.
Buchstaben-Fußnote (K)	Schlüsselwort. Definiert ein Schlüsselwort, das der Anwender für die Suche nach einem Thema verwenden kann. Schlüsselwörter sind optional.
Pluszeichen-Fußnote (+)	Nummer der Such-Sequenz. Definiert die Reihenfolge, in der der Anwender die Themen durchsuchen kann. Such-Sequenzen sind optional. Auch wenn Sie keine derartige Abfolge verwenden, ist der entsprechende Schalter vorhanden, hat dann jedoch keine Wirkung.
Durchgestrichener oder doppelt unterstrichener Text	Kreuzreferenz. Bezeichnet Text, den der Anwender für einen Sprung zu einem anderen Thema verwenden kann.
Unterstrichener Text	Definition. Legt fest, daß ein temporäres Fenster oder »Detailfenster« angezeigt werden soll, wenn der Anwender die Maustaste oder CR gedrückt hält. Das Fenster enthält Informationen, wie z.B. die Definition eines Wortes oder Ausdrucks, Hinweise zu einer Prozedur, etc.
Verborgener Text	Kreuzreferenz-Kontextstring. Bestimmt den Kontextstring für das Thema, das angezeigt wird, wenn der Anwender den unmittelbar vorausgehenden Text auswählt.

Wenn Sie symbolische Namen verwenden, müssen Sie diese am Anfang des Themas eintragen. Weitere Fußnoten können in beliebiger Reihenfolge angebracht werden. Informationen über die Zuweisung spezifischer Steuercodes finden Sie in den folgenden Abschnitten.

Symbolische Namen zuweisen

Symbolische Namen sind Strings, die einem Thema zuordnet werden und die festlegen, ob das betreffende Thema in die Hilfedatei aufgenommen wird oder nicht. Einem Thema können mehrere symbolische Namen zugewiesen werden. Sie sind jedoch nur optionale Bestandteile des Hilfesystems. Mit Hilfe dieser Markierungen können Sie verschiedene Versionen desselben Hilfesystems anlegen, ohne für jede Version eigene Quelldateien erstellen zu müssen. Themen ohne symbolische Namen werden immer in den Aufbau des Hilfesystems einbezogen.

Symbolische Namen werden als Fußnoten mit einem Stern angegeben. Wenn Sie einem Thema einen Namen zuweisen, schließt der Compiler das Thema in den Aufbau ein oder nicht, je nachdem, welche Festlegungen in der Option *Build* und dem [BuildTags]-Abschnitt der Hilfe-Projektdatei getroffen wurden. Im Abschnitt »Die Hilfedatei compilieren« finden Sie weitere Erläuterungen zu diesem Punkt. Zur Zuweisung eines symbolischen Namens müssen Sie folgende Arbeitsschritte ausführen:

1. Plazieren Sie den Cursor an den Anfang der ersten Zeile des Themas, so daß er vor allen nachfolgenden Fußnoten steht.

2. Geben Sie einen Stern (*) als Fußnotenmarkierung ein. Beachten Sie, daß der Stern hochgestellt (*) angezeigt wird.

3. Tippen Sie den symbolischen Namen als Fußnote ein. Zwischen dem Stern (*) und dem Namen darf nur ein Leerzeichen stehen.

Symbolische Namen können sich aus beliebigen alphanumerischen Zeichen zusammensetzen, wobei Groß-/Kleinschreibung keine Rolle spielt. Leerzeichen innerhalb des Wortes sind nicht erlaubt. Sie können mehrere symbolische Namen eingeben, die durch Strichpunkte voneinander abgetrennt werden müssen, z.B.:

```
* AppVersion1; AppVersion2; Test_Build
```

Die Angabe einer Namen-Fußnote bedeutet, daß der Name auf *True* gesetzt wird, wenn er mit dem Wert in der Projektdatei übereinstimmt. Der Compiler geht davon aus, daß alle anderen symbolischen Namen für dieses Thema den Wert *False* haben. Wenn der Name den Wert *True* erhalten hat, wertet der Compiler den *Build*-Ausdruck im [Options]-Abschnitt der Hilfe-Projektdatei aus. Beachten Sie, daß alle symbolischen Namen in der Projektdatei deklariert sein müssen, unabhängig davon, ob eine bedingte Compilierung diese deklariert oder nicht. Ergibt die Auswertung den Wert *True*, übernimmt der Compiler das Thema in den Aufbau der Hilfedatei. Andernfalls wird das Thema übergangen.

Der Compiler übernimmt alle Themen, die über keine symbolischen Namen verfügen, unabhängig davon, ob sie in der Hilfe-Projektdatei definiert sind oder nicht. Sie können deshalb symbolische Namen dazu verwenden, um gewisse Themen von bestimmten Compilierungen auszuschließen. Wenn der Compiler symbolische Namen antrifft, die nicht in der Hilfe-Projektdatei deklariert sind, gibt er eine Fehlermeldung aus.

Durch die Möglichkeit des Ein- bzw. Ausschließens bestimmter Themen können Sie mehrere unterschiedliche Hilfedateien aus denselben Themendatei-

en aufbauen. Dadurch sparen Sie Zeit und Arbeit bei der Entwicklung des Hilfesystems. Es bedeutet weiterhin, daß Sie für Ihre gesamte Produktlinie einen hohen Konsistenzgrad erreichen.

Kontextstrings zuweisen

Kontextstrings identifizieren die Themen im Hilfesystem. Ein Kontextstring muß eindeutig sein, d.h er darf in der Hilfe-Projektdatei nur einmal als Zuweisung zu einem Thema verwendet werden.

Über die Kontextstrings werden die Sprünge zwischen Themen zur Anzeige der Detailfenster realisiert. Die Zuweisung von Kontextstrings ist nicht obligatorisch. Dennoch sollten Sie den Themen Kontextstrings zuordnen, da Themen ohne diese Markierung nicht über Sprünge ansprechbar sind. Auf Themen ohne Kontextstring-Bezeichner wird über die Such-Sequenz oder über Schlüsselwörter zugegriffen. Die Entscheidung über die Vergabe von Kontextstrings liegt in der Verantwortung des Autors. Informationen über die Zuweisung von Sprüngen finden Sie im Abschnitt »Kreuzreferenzen zwischen Themen erstellen«. Die Zuweisung von Such-Sequenzen ist im Abschnitt »Nummern für die Funktion Durchsuchen zuweisen« beschrieben. Schlüsselwörter werden im Abschnitt »Schlüsselwörter zuweisen« weiter hinten in diesem Kapitel erläutert.

Führen Sie für die Zuweisung von Kontextstrings folgende Arbeitsschritte aus:

1. Plazieren Sie den Cursor vor den Anfang des Themas.

2. Geben Sie ein Nummernzeichen (#) als Fußnotenmarkierung ein. Beachten Sie, daß das Nummernzeichen hochgestellt ($^{\#}$) angezeigt wird.

3. Geben Sie den Kontextstring als Fußnote ein. Zwischen dem Nummernzeichen und dem String darf nur ein Leerzeichen stehen.

Folgende Zeichen sind in Kontextstrings zugelassen: alle Buchstaben von A-Z, alle Ziffern von 0-9, Punkte (.) und Unterstriche (_). Im folgenden sehen Sie die Kontextstring-Fußnote, die das Thema »Öffnen einer vorhandenen Textdatei« kennzeichnet:

```
#1ffnenVorhandeneTextdatei
```

Ein Kontextstring kann eine maximale Länge von 255 Zeichen haben, die Sie jedoch in der Praxis vernünftigerweise nicht ausnützen werden. Verwenden Sie kurze Kontextstrings, damit das Eintragen in die Textdateien möglichst einfach ist.

Titel zuweisen

Titel-Fußnoten führen folgende Aufgaben im Hilfesystem aus:

- Sie werden im *Lesezeichen*-Menü angezeigt.
- Sie werden in der Liste gefundener Einträge aufgeführt, die als Ergebnis einer Schlüsselwortsuche angezeigt wird (Themen ohne Titel, die über Schlüsselwörter ermittelt werden, sind als Themen ohne Titel in die Liste aufgenommen).

Titel sind nicht obligatorisch, obwohl Themen in den meisten Hilfesystemen einen Titel haben. Relativ unwichtige Themen, auf die die Suchfunktion, die Detailfenster und Systemmeldungen nicht zugreifen, benötigen keinen Titel.

Führen Sie für die Zuweisung von Titeln folgende Arbeitsschritte aus:

1. Plazieren Sie den Cursor vor den Anfang des Themas.

2. Geben Sie ein Dollarzeichen ($) als Fußnotenmarkierung ein. Beachten Sie, daß das Dollarzeichen hochgestellt ($^{\$}$) angezeigt wird.

3. Geben Sie den Titel als Fußnote ein. Zwischen dem Dollarzeichen und dem String darf nur ein Leerzeichen stehen.

Im folgenden sehen Sie eine Fußnote, die einen Titel für ein Thema definiert:

```
$ Hilfetasten
```

Bei der Vergabe von Titeln müssen Sie folgende Einschränkungen beachten:

Tabelle A.5 Einschränkungen bei Titeln

Art	Einschränkung
Zeichen	Titel können maximal 128 Zeichen lang sein. Längere Titel schneidet der Compiler ab. Der Hilfe-Compiler zeigt die Titel in einer Liste an, wenn der Anwender nach einem Schlüsselwort sucht oder ein Lesezeichen eingibt.
Formatierung	Titel-Fußnoten können nicht formatiert werden.

Schlüsselwörter zuweisen

Der Anwender kann nach Themen suchen, die mit einem Schlüsselwort versehen sind. Dabei vergleicht das Hilfesystem das vom Anwender eingegebene Schlüsselwort mit den Schlüsselwörtern, die einem bestimmten Thema zugewiesen sind. Das Hilfesystem listet dann die Titel der mit dem Schlüsselwort verknüpften Themen im Dialogfenster *Suchen* auf. Da der Anwender mit Schlüsselwörtern sehr schnell auf Themen zugreifen kann, sollten Sie den meisten Themen Schlüsselwörter zuweisen.

Hinweis: Sie können einem Thema nur dann ein Schlüsselwort zuweisen, wenn es über eine Titel-Fußnote verfügt. Dies ist notwendig, weil der Titel des Themas im Dialogfenster *Suchen* angezeigt wird, wenn der Anwender nach einem Schlüsselwort sucht.

Führen Sie für die Zuweisung von Titeln folgende Arbeitsschritte aus:

1. Plazieren Sie den Cursor vor den Anfang des Themas.

2. Geben Sie ein großes K als Fußnotenmarkierung ein. Beachten Sie, daß das K hochgestellt (^{K}) angezeigt wird.

3. Geben Sie den Titel als Fußnote ein. Zwischen K und dem ersten Schlüsselwort darf nur ein Leerzeichen stehen. Mehrere Schlüsselwörter werden durch Strichpunkte voneinander abgetrennt.

Im folgenden sehen Sie Schlüsselwörter für ein Thema:

```
K offen;öffnen;Textdatei;ASCII;vorhanden;nur Text;Dokumente;
```

Wenn der Anwender mit einem dieser Schlüsselwörter sucht, werden die Titel der Themen, die über die Schlüsselwörter verfügen, in einer Liste angezeigt. Ein Thema kann mehrere Schlüsselwörter haben. Bei der Vergabe von Schlüsselwörtern gelten folgende Einschränkungen:

Tabelle A.6 Einschränkungen bei Schlüsselwörtern

Art	Einschränkung
Zeichen	Für Schlüsselwörter sind alle ANSI-Zeichen zugelassen, einschließlich Akzente. Die maximale Länge für ein Schlüsselwort beträgt 255 Zeichen. Leerzeichen in Schlüsselwörtern werden als Zeichen interpretiert. Aus diesem Grund können Sie auch Sätze als Schlüsselwörter eingeben.
Sätze	Das Hilfesystem sucht nach jedem Wort des Satzes.
Formatierung	Schlüsselwörter sind unformatiert.
Groß-/Kleinschreibung	Die Groß-/Kleinschreibung spielt keine Rolle.
Satzzeichen	Mit Ausnahme des Strichpunkts, der als Trennzeichen dient, können alle Satzzeichen verwendet werden.

Zusätzliche Schlüsselworttabellen erstellen Mit Hilfe zusätzlicher Tabellen für Schlüsselwörter kann ein Anwender mit Hilfe des Programms Themen nachschlagen, die in anderen Schlüsselworttabellen stehen. Wenn Sie eine zusätzliche Schlüsselworttabelle verwenden, können Anwender Schlüsselwörter aus einer anderen, ihnen vertrauten Anwendung mit den Schlüsselwörtern in Ihrer Anwendung vergleichen.

Für die Erstellung zusätzlicher Tabellen müssen zwei Bedingungen erfüllt sein. Als erstes muß die Option *Multikey* in den [Options]-Abschnitt der Projektdatei eingebunden werden. Informationen zu dieser Option finden Sie im Abschnitt »Die Hilfedatei compilieren« unter »Zusätzliche Schlüsselworttabellen: Die Option Multikey«.

Zweitens müssen die Themen, die mit der zusätzlichen Tabelle verknüpft werden sollen, markiert werden. Die hierfür erforderlichen Fußnoten müssen mit dem in der Option *Multikey* angegebenen Buchstaben versehen werden. In diesem Fall ist die Groß-/Kleinschreibung der Schlüsselwort-Fußnote zu berücksichtigen. Sie müssen also darauf achten, daß Sie immer dieselbe Schreibweise für den Kennzeichnungsbuchstaben verwenden (normalerweise Großschreibung). Das Hilfesystem zeigt bei einer Suche mit mehreren Tabellen nicht das normale Dialogfenster *Suchen* an. Es wird stattdessen das erste Thema aufgeführt, das das angegebene Schlüsselwort enthält. Wenn die Schlüsselwörter aus zusätzlichen Tabellen in der regulären Schlüsselworttabelle erscheinen sollen, müssen Sie dem betreffenden Schlüsselwort ein *K* voranstellen.

Die Anwendung kann nun das Hilfethema, das mit dem gegebenen String in einer bestimmten Schlüsselworttabelle verknüpft ist, anzeigen. Schlüsselwörter werden für die Tabelle ohne Berücksichtigung der Groß-/Kleinschreibung sortiert. Information über die Parameter, die eine Anwendung beim Aufruf eines

Themas in einer zusätzlichen Schlüsselworttabelle übergibt, finden Sie im Abschnitt »Die Hilfedatei compilieren«.

Nummern für die Funktion Durchsuchen zuweisen

Hinweis: In dieser Version des Hilfesystems wird auf die Themen über die *Durchsuchen*-Schalter in der obersten Zeile des Hilfefensters zugegriffen. In künftigen Versionen werden diese Schalter nur noch dann angezeigt, wenn Ihre Hilfe-Ressourcendatei eine Such-Sequenz enthält.

Mit den Schaltern *Durchsuchen* >>und *Durchsuchen* <<in der Symbolzeile des Hilfefensters kann der Anwender verwandte Themen vorwärts und rückwärts durchsuchen. Die Reihenfolge, in der Themen durchsucht werden, ist durch die »Such-Sequenz« festgelegt. Die Abfolge der Themen wird durch Nummern bestimmt, die der Autor vergibt.

Folgende Arbeitsschritte müssen für den Aufbau einer Such-Sequenz ausgeführt werden:

1. Festlegung der Themen, die zusammengefaßt werden sollen und Bestimmung der Reihenfolge, in der die einzelnen Themen bei Anzeige der gesamten Gruppe erscheinen sollen. Das Hilfesystem unterstützt zur Festlegung der Reihenfolge mehrere, nicht zusammengehörige Listen.

2. Codierung der Themen für die Implementierung der Such-Sequenz.

Elemente von Such-Sequenzen anordnen Bei der Organisation der Such-Sequenzen muß der Autor darauf achten, daß die Abfolge der Themen sinnvoll ist. Innerhalb eines Themenkreises können die einzelnen Themen in alphabetischer Reihenfolge, entsprechend ihrem Schwierigkeitsgrad oder gemäß der Anwendungslogik angeordnet werden. Im folgenden Beispiel sehen Sie Such-Sequenzen für die Menübefehle einer gegebenen Anwendung. Der Autor hat die Reihenfolge so festgelegt, wie sie ihm unter prozeduralen Aspekten sinnvoll erschien. Es sind aber selbstverständlich andere Anordnungen denkbar.

```
SampleApp Commands
   File Menu - commands:005
     New Command - file_menu:005
     Open Command - file_menu:010
     Save Command - file_menu:015
     Save As Command - file_menu:020
     Print Command - file_menu:025
     Printer Setup Command - file_menu:030
     Exit Command - file_menu:035
   Edit Menu - commands:010
     Undo Command - edit_menu:025
     Cut Command - edit_menu:015
     Copy Command - edit_menu:010
     Paste Command - edit_menu:020
     Clear Command - edit_menu:005
     Select All Command - edit_menu:030
     Word Wrap Command - edit_menu:035
     Type Face Command - edit_menu:040
```

```
    Point Size Command - edit_menu:045
  Search Menu - commands:015
    Find Command - search_menu:005
    Find Next Command - search_menu:010
    Previous Command - search_menu:015
  Window Menu - commands:020
    Tile Command - window_menu:005
    Cascade Command - window_menu:010
    Arrange Icons Command - window_menu:015
    Close All Command - window_menu:020
    Document Names Command - window_menu:025
```

Jede Zeile enthält einen Sequenzlistennamen, auf den ein Doppelpunkt und eine Sequenznummer folgen. Die Angabe eines Sequenzlistennamens ist optional. Hat ein Element einer Sequenz keinen Listennamen – wie im folgenden Beispiel – plaziert der Compiler das Thema in eine »Null«-Liste:

```
Window Menu - 120
```

Die Nummern im obigen Beispiel beginnen bei 005 und werden um jeweils 005 hochgezählt. Es ist ratsam, bei der Numerierung größere Zahlenschritte zu vergeben, damit später neue Themen eingefügt werden können (falls erforderlich). Ausgelassene Zahlen bei der Numerierung spielen für den Hilfe-Compiler keine Rolle. Es ist nur wichtig, daß die Reihenfolge aufsteigend ist.

Sequenznummern legen die Reihenfolge fest, in der die Themen in einer Such-Sequenz angeordnet werden. Für die Nummern können beliebige alphanumerische Zeichen vergeben werden. Während der Compilierung werden die Strings nach der ASCII-Abfolge sortiert, nicht nach einer numerischen Sortierordnung.

Die alphanumerischen und numerischen Bestandteile einer Sequenz können unterschiedlich lang sein; jedoch sollte die einmal vergebene Länge in allen Themendateien beibehalten werden. Wenn Sie nur Zahlen in den Strings einsetzen, müssen alle Strings gleich lang sein. Andernfalls könnte eine höhere Sequenznummer unter bestimmten Umständen vor einer niedrigeren Nummer stehen. So ist beispielsweise 100 numerisch gesehen höher als 99. 100 erscheint aber in der vom Hilfe-Compiler verwendeten Sortierordnung vor 99, weil der Hilfe-Compiler die beiden ersten Stellen im String vergleicht. Aus diesem Grund müssen Sie anstelle von `99` die Zahl `099` eingeben.

Such-Sequenzen codieren Wenn Sie die Gruppierung und Anordnung der Themen festgelegt haben, codieren Sie die Sequenz, indem Sie jedem Thema einen Sequenzlistennamen und eine Nummer zuweisen.

Führen Sie dazu die folgenden Arbeitsschritte aus:

1. Stellen Sie den Cursor an den Beginn des Themas.

2. Geben Sie ein Pluszeichen (+) als Fußnotenmarkierung ein. Beachten Sie, daß das Pluszeichen hochgestellt ($^{+}$) angezeigt wird.

3. Geben Sie die Sequenznummer ein. Dazu stehen Ihnen alle alphanumerischen Zeichen zur Verfügung.

Die folgende Fußnote definiert zum Beispiel die Such-Sequenznummer für das Thema Edit-Menü in der vorausgehenden Such-Sequenz:

```
+ commands:010
```

Sie müssen die Themen in der Datei nicht in der Reihenfolge eingeben, in der sie in der Such-Sequenz erscheinen sollen (obwohl dies einfacher ist), da der Compiler diese Aufgabe für Sie übernimmt.

Kreuzreferenzen zwischen Themen erstellen

Kreuzreferenzen oder »Sprünge« sind speziell codierte Wörter oder Sätze, die zwei Themen miteinander verknüpfen. Sprungbegriffe werden in der Themendatei als durchgestrichener oder doppelt unterstrichener Text markiert. Im Hilfefenster werden diese Begriffe dann unterstrichen angezeigt. Beispielsweise erscheinen die durchgestrichenen (bzw. in Word für Windows doppelt unterstrichenen) Wörter ~~Neue Befehle~~ als Neue Befehle im Hilfefenster.

Führen Sie folgende Arbeitsschritte aus, um ein Wort oder einen Satz als Sprungstelle zu einem anderen Thema zu definieren:

1. Stellen Sie den Cursor an die Stelle im Text, an der Sie den Sprungbegriff eingeben wollen.

2. Wählen Sie die Textprogrammoption für Durchstreichung oder Doppelte Unterstreichung an.

3. Geben Sie das Sprungwort oder den Sprungsatz als durchgestrichenen Text ein.

4. Schalten Sie die Durchstreichung wieder aus und wählen Sie die Option zum Verbergen von Text an.

5. Geben Sie den Kontextstring ein, der mit dem Thema verknüpft ist, zu dem der Sprung erfolgen soll.

Bei der Codierung von Sprüngen gelten folgende Regeln:

- Zwischen dem durchgestrichenen (oder doppelt unterstrichenen) Text und dem verborgenen Text darf kein Leerzeichen stehen.
- Vor oder nach dem Sprungbegriff darf kein Leerzeichen angegeben werden.
- Absatzmarkierungen müssen als Text eingegeben werden.

Begriffe definieren

Die meisten Themendateien enthalten Wörter oder Sätze, die einer weiteren Definition bedürfen. Wenn der Anwender eine Begriffsdefinition haben möchte, muß er zuerst das gewünschte Wort auswählen und dann den Mausknopf gedrückt halten oder ↵ betätigen: Die Definition wird in einem zusätzlichen Fenster (Detailfenster) im Hilfefenster eingeblendet. Der Autor entscheidet, für welche Wörter Definitionen erforderlich sind, wobei der Anwenderkreis der Anwendung berücksichtigt werden muß.

Hinweis: Die Anzeige eines Detailfensters ist aber nicht auf Definitionen beschränkt. Sie können die Möglichkeit einer zeitlich begrenzten Anzeige von Informationen zum Beispiel auch für Hinweise zu einer Prozedur ausnutzen.

Führen Sie zur Definition von Begriffen folgende Arbeitsschritte aus:

- Erfassen Sie ein Thema, das den Begriff definiert. Das Definitionsthema muß einen Kontextstring beinhalten (vgl. den Abschnitt »Kontextstrings zuweisen« weiter vorne in diesem Kapitel).

- Legen Sie Kreuzreferenzen für alle Belegstellen des Definitionsbegriffs an. Sie müssen die Codierung für den Sprung jedoch nur bei dem ersten Auftreten des Begriffs anbringen. Bedenken Sie auch den Umfang von farbig hervorgehobenem Text im Hilfefenster (vgl. den Abschnitt »Definitionen codieren« weiter hinten in diesem Kapitel.)

Definitionsthemen codieren Sie können die Themen, die die Definitionen enthalten, beliebig anordnen. So können die Definitionsthemen in der Themendatei speichern, in denen die entsprechenden definierten Begriffe stehen. Sie können aber auch alle Definitionsthemen in einer Datei zusammenfassen und den Anwender direkt darauf zugreifen lassen.

Bei Helpex wird letztere Methode verwendet. Die Definitionen stehen alle in der Datei TERMS.RTF. Wenn Sie alle Definitionen in eine Datei schreiben, haben Sie ein Glossar zur Verfügung, das sich sehr leicht ändern läßt.

Definitionen codieren Wenn Sie Definitionsthemen erfaßt haben, müssen Sie diese auf folgende Art und Weise codieren:

1. Bestimmen Sie die Stelle, an der der Begriff, der eine Definitionen erfordert, stehen soll.

2. Wählen Sie die Unterstreichen-Option Ihres Textprogramms an.

3. Geben Sie den Begriff ein.

4. Schalten Sie die Unterstreichung wieder aus und wählen Sie die Option zum Verbergen von Text an.

5. Geben Sie den Kontextstring ein, der zu dem Thema gehört, das den Definitionsbegriff enthält.

Grafikbilder einbringen

Bitmap-Grafiken können auf zweierlei Art und Weise in das Hilfesystem eingebracht werden. Wenn Ihr Textprogramm das direkte Einfügen von Grafiken der Windows-Versionen 2.1 oder 3.0 unterstützt, können Sie die Bitmaps in jede Themendatei einfügen.

Sie können – als zweite Möglichkeit – jede Bitmap-Grafik in einer separaten Datei speichern und an der Stelle, an der die Grafik in der Hilfe-Themendatei plaziert werden soll, über den Dateinamen eine Referenz auf die Bitmap-Datei angeben. Letztere Methode nennen wir »referenzierte Bitmaps«.

In den folgenden Abschnitten werden beide Prozesse zum Einbinden von Grafiken erläutert.

Bitmaps anlegen und einbinden

Sie können Bitmaps mit jedem Grafikprogramm anlegen, dessen Bilder in der Windows-Umgebung angezeigt werden können. Die Bitmaps werden in die Zwischenablage von Windows kopiert. Von dort können sie in ein Grafikbearbeitungsprogramm, wie beispielsweise Paint, kopiert und – falls gewünscht – geändert werden.

Windows Help 3.0 unterstützt farbige Bitmaps. Es ist aber sinnvoll, wenn Sie die Bitmaps unter dem Aspekt der weiteren Verwendbarkeit auf Schwarzweiß-Darstellung beschränken. Wenn Sie sich für die Erstellung von Schwarzweiß-Grafiken entscheiden, müssen Sie die ursprünglich farbige Grafik manuell an die Schwarzweiß-Gegebenheiten anpassen.

Sobald Sie mit dem Erscheinungsbild Ihrer Grafik zufrieden sind, können Sie diese in einer Datei abspeichern (»referenziertes Bitmap«) oder sie in die Zwischenablage kopieren und von dort in Ihr Bearbeitungsprogramm kopieren. Beim Abspeichern in eine Datei müssen Sie als erstes dem Grafikbearbeitungsprogramm die Größe des Bitmaps angeben, damit nur der für die Anzeige im Hilfefenster relevante Teil des Bildes gesichert wird. Speichern Sie nur soviel von der Grafik ab, wie tatsächlich im Hilfefenster angezeigt werden soll. Denn wenn die Grafik klein genug ist, haben Sie die Möglichkeit, Text neben dem Bild anzubringen. Speichern (oder konvertieren und speichern) Sie Grafiken immer im Windows-Format .BMP.

Bitmaps sollten in demselben Bildschirmmodus angelegt werden, in dem das Hilfesystem die Bilder anzeigen soll. Wenn die Themendateien in einem anderen Modus angezeigt werden, entspricht die Darstellung der Bitmaps unter Umständen nicht den Originalgrafiken.

Bitmaps mit einem Grafikbearbeitungsprogramm plazieren

Mit Hilfe eines Textprogramms mit grafischer Oberfläche können Bitmaps auf sehr einfache Art und Weise in das Hilfesystem eingebracht werden. Microsoft Word für Windows unterstützt den direkten Import von Bitmaps aus der Zwischenablage. Bringen Sie die Grafik aus der Zwischenablage einfach an die im Hilfethema gewünschte Position. Sie können den Text so formatieren, daß er unterhalb oder neben der Grafik steht. Wenn Sie die Themendatei in dem RTF-Dateiformat speichern, wird das importierte Bitmap ebenfalls konvertiert und automatisch in die Hilfe-Ressourcendatei eingebunden.

Referenzierte Bitmaps

Wenn Ihr Textprogramm Grafiken nicht direkt importieren und anzeigen kann, geben Sie die Datei an, in der das Bitmap gespeichert ist. Zur Angabe einer Referenz auf eine Grafikdatei in der Themendatei müssen Sie eine der folgenden Anweisungen eingeben. Fügen Sie die Anweisung an der Stelle ein, an der das Bild im Thema erscheinen soll:

```
{bmc Dateiname.bmp}
{bml Dateiname.bmp}
{bmr Dateiname.bmp}
```

Hinweis: Geben Sie für *Dateiname* nicht den vollen Pfad an. Wenn der Compiler eine Grafik in einem anderen als dem Hauptverzeichnis suchen soll,

müssen Sie den vollständigen Pfadnamen in dem [Bitmaps]-Abschnitt der Projektdatei angeben.

Das Argument *bmc* steht für »bitmap character« und zeigt an, daß das betreffende Bitmap wie ein Zeichen behandelt wird, das in der Themendatei an derselben Position in einer Zeile steht. In derselben Zeile kann Text vor oder nach dem Bitmap stehen. Der Zeilenabstand wird durch die Größe der Zeichen (einschließlich des Bitmap-Zeichens) bestimmt. Geben Sie keinen negativen Zeilenabstand zwischen Absätzen mit Grafiken an. Dadurch könnte das Bild den über ihm stehenden Text bei der Anzeige im Hilfefenster überschreiben. Bei Verwendung des Arguments *bmc* wird Text nicht automatisch um das Bild gruppiert, sondern folgt nach dem Bild auf der Grundlinie.

Das Argument *bml* legt fest, daß das Bild am linken Rand angezeigt wird und daß der Text rechts neben dem Bild steht. Das Argument *bmr* plaziert das Bild an den rechten Rand und den Text links daneben. Die Dateinamen für die Bitmaps müssen dieselben sein, die in dem [Bitmaps]-Abschnitt der Projektdatei stehen. Der [Bitmaps]-Abschnitt wird im Abschnitt »Die Hilfedatei compilieren« erläutert.

Hinweis: Mehrere Referenzen auf ein Bitmap mit demselben Namen beziehen sich bei der Anzeige der Hilfedatei auf dieselbe Grafik. Das bedeutet, daß die Bitmap-Referenzen im Hilfesystem wiederholt werden können, ohne daß dadurch die Ressourcendatei wesentlich vergrößert wird.

In Abbildung A.6 sehen Sie, wie ein Thema bestehend aus Text und drei Grafiken, im Windows-Hilfesystem dargestellt wird.

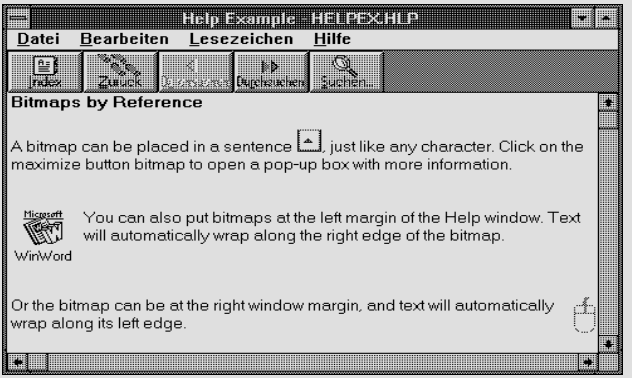


Abbildung A.6 Anzeige eines Hilfethemas mit referenzierten Grafiken

Themendateien verwalten

Themendateien können im Standardformat des Textprogramms oder im RTF-Format abgespeichert werden. Wenn Sie Ihre Dateien immer im RTF-Format speichern, und später Änderungen vornehmen, benötigt das Textprogramm zusätzliche Zeit zur Interpretation des Formats, wenn die Datei erneut geladen wird. Wenn Sie davon ausgehen, daß Sie bei der Entwicklung des Hilfesystems häufig Änderungen vornehmen werden, können Sie das Zeit-

verhalten optimieren, indem Sie die Dateien im Standard- und RTF-Format abspeichern. Vergeben Sie zur Unterscheidung der Dateien verschiedene Dateinamenserweiterungen. Der Compiler benötigt nur die RTF-Dateien und Sie haben für Änderungen schnelleren Zugriff auf die Quelldateien. Bei einem umfangreichen Projekt können Sie so spürbar Zeit bei der Entwicklung einsparen.

Dateien und Themen überwachen

Aus folgenden Gründen ist es sehr wichtig, daß Sie die immer einen Überblick über die Themen und Dateien haben:

- Sie müssen sichergehen, daß alle Themen in die Compilierung einbezogen werden.
- Jedem Thema muß ein eindeutiger Kontextstring zugewiesen sein.
- Such-Sequenzen müssen in allgemeinen und speziellen Listen überprüft werden.
- Schlüsselwort- und Titelabgleich muß angezeigt werden.
- Der Autor muß sehen können, an welcher Stelle der Text für die Themen steht.
- Dateiänderungen und der aktuelle Status müssen protokolliert werden.

Der Autor muß zumindest seine eigenen Themendateien überwachen und die Dateinamen an denjenigen Mitarbeiter weitergeben, der für die Erstellung der Projektdatei verantwortlich ist.

Hilfeprotokoll anlegen

Das Überwachungs-Tool kann – neben der Protokollierung der am Entwicklungszyklus beteiligten Dateien – weitere Aufgaben erfüllen. Sie können eine aktuelle Liste der Themen in einer ASCII-Textdatei, einem Quattro-Pro-Arbeitsblatt oder einem anderen Format mitführen.

Wenn Sie oder ein anderer Mitarbeiter ein Thema anlegen oder überarbeiten, aktualisiert das Hilfeprotokoll die davon betroffene Datei. Die Inhalte der Protokolldatei folgen keiner strengen Definition. Eine Protokolldatei sollte aber Dateinamen, Kontextstring, Titel, Such-Sequenz und Schlüsselwörter enthalten. Wenn Ihr Hilfesystem kontextabhängig aufgebaut ist, sollten auch diese Zusammenhänge überwacht werden. Dieser Eintrag ist nur dann notwendig, wenn Sie den Themen in der Projektdatei Kontextnummern zuweisen. Wenn Sie einen Gesamtüberblick über das Hilfesystem erstellen wollen, können Sie noch weitere Informationen aufnehmen, wie Erstellungsdatum, Änderungsdatum, Status, Verfasser. Die Anordnung und Aufnahme von Informationen liegt allein in Ihrem Ermessen.

Das in folgender Abbildung dargestellte Beispiel für ein Arbeitsblattprotokoll zeigt, wie Themen des Hilfesystems angeordnet werden könnten. Das Beispiel enthält sowohl das Hilfemenü als auch kontextabhängige Einträge für die Themendateien. Auf die Themen, die der Anwender über die Auswahl von Befehlen im Hilfemenü erreichen kann, kann ebenfalls über die Kontextabhängigkeit zugegriffen werden. Die Themen mit Einträgen in der Kontext-ID-Spalte

werden für die Kontextabhängigkeit und für die Menübefehle verwendet. Beachten Sie, daß einige Themen mehrere Nummern für Kontextabhängigkeiten haben.

Dadurch kann das Thema, wenn es vom Anwender angeklickt wird, in unterschiedlichen Bildschirmbereichen angezeigt werden.

	A	B	C	D	E	F	G	H
1	**Ctx. String**	**Title**	**Browse Seq.**	**Key Words**	**Ctx. No.**	**Filename**	**Modified**	**Status**
2	hipidx_idx_mp	Multipad Help Index			0xFFFF	helpex.idx	5/16/89	Done
3	mc_cmd_mp	Multipad Commands	commands:0001	commands; menus	0x1000	helpex.cmd	5/16/89	Done
4	fm_cmd_mp	File Menu	commands:0004	commands; menus; files;	0x1001	helpex.cmd	5/16/89	Done
5				documents				
36	lfc_cmd_mp	Type Face Command	commands:0068	commands; edit editing; type face;	0x1011	helpex.cmd	5/16/89	Test
37				font; Text; format				
38	ptsc_cmd_mp	Point Size Command	commands:0072	commands; edit; editing; point size;	0x1012	helpex.cmd	5/16/89	Test
39				text, format				

Abbildung A.7 Hilfeprotokoll als Arbeitsblatt

Die Hilfedatei compilieren

Nachdem Sie die Themendateien für Ihr Hilfesystem erstellt haben, sind Sie soweit, eine Hilfe-Projektdatei zu erstellen und damit eine erste Test-Compilierung Ihrer Hilfedatei durchzuführen. Die Hilfe-Projektdatei enthält alle Informationen, die der Compiler für die Umwandlung der Themendateien in eine binäre Hilfe-Ressourcendatei benötigt.

Die Hilfe-Projektdatei teilt dem Compiler mit, welche Themendateien in die Compilierung einbezogen werden müssen. Sie enthält außerdem Informationen, die dem Compiler den Vergleich spezieller Themen mit den Kontextnummern ermöglichen (dies ist für den kontextabhängigen Teil des Hilfesystems erforderlich).

Das Entwicklerteam des Anwenderprogramms hat im Anschluß an die Compilierung dafür zu sorgen, daß der Datenaustausch zwischen Anwendung und Hilfesystem reibungslos und fehlerfrei erfolgen kann.

Im vorliegenden Abschnitt werden folgende Punkte behandelt:

- Erstellen einer Hilfe-Projektdatei;
- Compilieren der Hilfedatei;
- Programmieren der Anwendung für einen Zugriff auf das Hilfesystem.

Hilfe-Projektdatei erstellen

Mit der Hilfe-Projektdatei steuern Sie, wie der Hilfe-Compiler mit Ihren Themendateien umgeht.

Die Hilfe-Projektdatei kann bis zu sechs Abschnitte enthalten, die die folgenden Funktionen ausführen:

Tabelle A.7 Abschnitte in der Projektdatei

Abschnitt	Funktion
[Files]	Legt fest, welche Themendateien beim Compilieren (Build) berücksichtigt werden. Dieser Abschnitt ist obligatorisch.
[Options]	Bestimmt, welche Fehlermeldungen angezeigt werden, welche Themen in die Compilierung einbezogen werden, in welchem Verzeichnis sich die Dateien und der Hilfeindex befinden. Dieser Abschnitt ist optional.
[BuildTags]	Spezifiziert die gültigen symbolischen Namen. Dieser Abschnitt ist optional.
[Alias]	Weist einem Thema einen oder mehrere Kontextstrings zu. Dieser Abschnitt ist optional.
[Map]	Verknüpft die Kontextstrings mit einer Kontextnummer. Dieser Abschnitt ist optional.
[Bitmaps]	Gibt an, welche Bitmap-Dateien in die Compilierung einbezogen werden. Dieser Abschnitt ist optional.

Die Hilfe-Projektdatei kann mit jedem ASCII-Textprogramm erstellt werden. Hilfe-Projektdateien haben die Namenserweiterung .HPJ. Wenn Sie die .HPJ-Erweiterung in der HC-Befehlszeile nicht angeben, sucht der Compiler vor dem Laden der Projektdatei nach einer Datei mit dieser Erweiterung. Die .HLP-Ausgabedatei erhält den gleichen Namen wie die .HPJ-Datei.

Die Reihenfolge der oben aufgeführten Abschnitte innerhalb der Hilfe-Projektdatei kann, mit einer Ausnahme, willkürlich gewählt werden.

Die Ausnahme bildet der [Alias]-Abschnitt: Dieser muß (falls er verwendet wird) unter allen Umständen vor dem [Map]-Abschnitt stehen. Die Namen der Abschnitte müssen in eckigen Klammern stehen.

Verwenden Sie folgendes Format:

[*Abschnittname*]

Kommentare in der Projektdatei können durch Strichpunkte gekennzeichnet werden. Der Compiler ignoriert Text, der zwischen einem Strichpunkt und dem Zeilenende steht.

Themendateien festlegen: Der [Files]-Abschnitt

Im [Files]-Abschnitt der Hilfe-Projektdatei werden alle Themendateien aufgelistet, die der Hilfe-Compiler für die Erstellung der Hilfe-Ressource verwenden soll. Jede Hilfe-Projektdatei *muß* einen [Files]-Abschnitt enthalten. Das folgende Beispiel zeigt das Format des [Files]-Abschnitts:

```
[FILES]
HELPEX.RTF  ;Hauptthemen für die Anwendung HelpEx
TERMS.RTF   ;Begriffe für die Anwendung HelpEx
```

Der Hilfe-Compiler sucht in dem in der Option *Root* definierten Verzeichnis nach den Dateien, die in diesem Abschnitt der Hilfe-Projektdatei aufgeführt sind. Befindet sich eine angegebene Datei nicht in diesem Verzeichnis, erzeugt der Hilfe-Compiler einen Fehler. Näheres zur Option *Root* erfahren Sie im Abschnitt »Arbeitsverzeichnis angeben: Die Root-Option« weiter hinten in diesem Kapitel.

Mit der aus der Sprache C bekannten `#include` -Anweisung können Sie Dateien, die weitere Anweisungen zur Compilierung der Hilfedatei enthalten, in die Projektdatei einbinden:

```
#include <Dateiname>
```

Die Zeile muß mit einem Nummernzeichen (#) beginnen, und der Dateiname muß in spitzen Klammern stehen. Geben Sie als Dateinamen einen vollständigen Pfad ein (entweder den in der Option *Root* definierten Pfad oder einen absoluten Verzeichnispfad).

Meist ist es am einfachsten, die Dateien für das Hilfeprojekt in einer gesonderten Textdatei aufzuführen und diese Textdatei dann bei der Compilierung über die `#include` -Anweisung in die Projektdatei einzubinden:

```
[FILES]
#include <hlpfiles.inc>
```

Symbolische Namen festlegen: Der [BuildTags]-Abschnitt

Enthalten Ihre Themendateien symbolische Namen, können Sie diese im Abschnitt [BuildTags] der Hilfe-Projektdatei angeben. Die Erstellung eines [BuildTags]-Abschnitts ist optional. Das folgende Beispiel zeigt das Format des [BuildTags]-Abschnitts einer Hilfe-Projektdatei:

```
[BUILDTAGS]
WINENV      ;topics to include in Windows build
DEBUGBUILD  ;topics to include in debugging build
TESTBUILD   ;topics to include in a mini-build for testing
```

Der [BuildTags]-Abschnitt kann maximal 30 symbolische Namen enthalten. Bei symbolischen Namen spielt die Groß-/Kleinschreibung keine Rolle. Leerzei-

chen dürfen in symbolischen Namen nicht enthalten sein. Im [BuildTags]-Abschnitt darf nach einem symbolischen Namen nur ein Kommentar folgen – andernfalls erzeugt der Compiler eine Fehlermeldung.

Die Angabe symbolischer Namen in Themendateien wird unter »Symbolische Namen zuweisen« im Abschnitt »Themendateien erstellen« beschrieben.

Optionen festlegen: Der [Options]-Abschnitt

Im Abschnitt [Options] der Hilfe-Projektdatei können Sie folgende Optionen angeben:

Tabelle A.8 [Options]-Optionen

Option	Bedeutung
BUILD	Legt die Themen für den Compilier-Vorgang fest.
COMPRESS	Komprimiert die Hilfe-Ressourcendatei.
FORCEFONT	Erzeugt eine Hilfe-Ressourcendatei mit nur einer Schrift.
INDEX	Spezifiziert den Kontextstring für den Hilfeindex.
MAPFONTSIZE	Gibt an, daß bestimmte Schriftgrößen umgewandelt werden sollen.
MULTIKEY	Spezifiziert einen Vergleich zwischen zusätzlichen Schlüsselwort-tabellen.
ROOT	Legt das Arbeitsverzeichnis für die Compilierung fest.
TITLE	Spezifiziert den Titel für das Hilfesystem.
WARNING	Bestimmt ab welchem Level der Compiler Fehlermeldungen ausgibt.

Die Reihenfolge, in der diese Optionen im [Options]-Abschnitt erscheinen, kann willkürlich gewählt werden. Die Definition eines [Options]-Abschnitts ist nicht obligatorisch. In den folgenden Abschnitten wird ausführlich auf die einzelnen Optionen eingegangen.

Fehlermeldungen ausgeben: Die Warning-Option

Mit der Option *Warning* legen Sie fest, ab welcher Fehlerebene der Compiler eine Meldung ausgibt. Die Syntax für *Warning* lautet:

WARNING=Ebene

Sie können zwischen drei Fehlerebenen wählen:

Tabelle A.9 WARNING-Ebenen

Ebene	Art des Fehlers
1	Schwere Fehler.
2	Mittelschwere und schwere Fehler.
3	Alle Fehler. Dies ist die Voreinstellung, wenn die Option *Warning* nicht angegeben wird.

Der folgende Code weist den Compiler an, für mittelschwere und schwere Fehler eine Meldung auszugeben:

```
[OPTIONS]
WARNING=2
```

Der Compiler übergibt die Fehlermeldungen an das Standardausgabegerät (normalerweise den Bildschirm). Sie können die Ausgabe allerdings auch in eine Datei umleiten und sich diese beim Austesten des Hilfesystems ansehen.

Das folgende Beispiel leitet die Bildschirmausgabe des Compilers in eine Datei um:

```
HC HELPEX > errors.out
```

Hinweis: Wenn Sie vor Beginn der Compilierung Strg Druck drücken, werden die Fehlermeldungen auf dem Bildschirm und auf dem Drucker ausgegeben. Erneutes Drücken von Strg Druck beendet das Senden der Meldungen an den Drucker.

Themen mit symbolischen Namen angeben: Die Build-Option

Enthalten Ihre Themendateien symbolische Namen, können Sie mit der *Build*-Option festlegen, welche Themen in die Compilierung einbezogen werden sollen. Verwenden Sie diese Option nicht, wenn Ihre Themendateien keine symbolischen Namen enthalten.

Hinweis: Im [BuildTags]-Abschnitt der Projektdatei müssen alle symbolischen Namen aufgelistet werden, gleichgültig, ob die spätere bedingte Compilierung die Namen deklariert.

Im Abschnitt »Themendateien erstellen« wird ausführlich beschrieben, wie den Themen symbolische Namen zugewiesen werden. Die Syntax für *Build* lautet:

BUILD=Ausdruck

Build-Ausdrücke dürfen maximal 255 Zeichen lang sein und müssen in einer Zeile stehen. Für *Build*-Ausdrücke führt der Compiler eine Boolesche Auswertung durch. Die Ergebnisse dieser Auswertungen entscheiden, welche Themen der angegebenen Themendateien vom Compiler in die Hilfedatei aufgenommen werden.

Der Compiler wertet alle *Build*-Ausdrücke von links nach rechts aus. Folgende, gemäß ihrer Priorität in absteigender Reihenfolge aufgeführten Sprachelemente sind zulässig:

Tabelle A.10 Reihenfolge der Build-Ausdrücke

Ausdruck	Beschreibung
<tag>	Symbolischer Name
( )	Klammern
~	NICHT-Operator
&	UND-Operator
|	ODER-Operator

Angenommen, Ihre Themendateien enthalten die symbolischen Namen WINENV, APP1 und TEST_BUILD. Sie könnten dann einen der folgenden *Build*-Ausdrücke in den [Options]-Abschnitt aufnehmen:

Tabelle A.11 Beispiele für Build-Ausdrücke

BUILD-Ausdruck	Berücksichtigte Themen
BUILD=WINENV	Nur Themen mit dem Namen WINENV.
BUILD=WINENV&APP1	Themen, die sowohl den Namen WINENV als auch den Namen APP1 haben.
BUILD=WINENV|APP1	Themen, die entweder den Namen WINENV oder den Namen APP1 haben.
BUILD=(WINENV|APP1) &TESTBUILD	Themen, die entweder den Namen WINENV oder den Namen APP1 und zusätzlich den Namen TESTBUILD haben.
BUILD=~APP1	Themen, die nicht den Namen APP1 haben.

Arbeitsverzeichnis angeben: Die Root-Option

Mit der Option *Root* geben Sie das Arbeitsverzeichnis für das Hilfeprojekt an. Der Compiler sucht in diesem Verzeichnis nach den Dateien. Die Syntax für *Root* lautet:

ROOT=Pfadname

Die folgende *Root*-Option setzt das Arbeitsverzeichnis auf \BUILD\TEST in Laufwerk D:

```
[OPTIONS]
ROOT=D:\BUILD\TEST
```

Mit Hilfe der Option *Root* können Sie Pfade angeben, die sich auf das Arbeitsverzeichnis des Hilfeprojekts beziehen. Der folgende Eintrag im [Files]-Abschnitt bezieht sich z.B. auf einen relativen Pfad, ausgehend vom Arbeitsverzeichnis:

```
TOPICS\FILE.RTF
```

Wenn Sie sich auf eine Datei in einem bestimmten, vom Arbeitsverzeichnis des Hilfeprojekts unabhängigen Verzeichnis beziehen wollen, müssen Sie einen vollständigen (absoluten) Pfad angeben, in dem auch (falls nötig) der Laufwerksbuchstabe enthalten ist:

```
D:\HELPTEST\TESTFILE.RTF
```

Wenn Sie die *Root*-Option für Ihr Hilfeprojekt nicht verwenden, beziehen Sie alle Pfadangaben auf das aktuelle DOS-Verzeichnis.

Index festlegen: Die Index-Option

Mit der Option *Index* können Sie den Kontextstring für den Hilfeindex definieren. Da der Anwender mit dem Index-Schalter von jeder Stelle des Hilfesystems aus auf den Index zugreifen kann, werden Sie wahrscheinlich keine Sprungbegriffe für den Index schreiben wollen.

Der Zugriff auf diesen allgemeinen Index erfolgt dann entweder über das

Hilfemenü der Anwendung oder durch die Auswahl des Index-Schalters im Hilfefenster.

Durch die Zuweisung eines Kontextstrings an das Indexthema im [Options]-Abschnitt teilen Sie dem Compiler die Position des Hauptindex der Hilfethemen in der Hilfedatei mit. Wenn Sie die *Index*-Option im [Options]-Abschnitt nicht verwenden, verwendet der Compiler das erste angetroffene Thema als Index. Die Syntax für *Index* lautet:

INDEX=Kontextstring

Der angegebene Kontextstring muß mit demjenigen übereinstimmen, den Sie dem Hilfeindexthema zugewiesen haben.

Das folgende Beispiel teilt dem Compiler mit, daß der Kontextstring für den Hilfeindex *main_index* lautet:

```
[OPTIONS]
INDEX=main_index
```

Die Zuweisung von Kontextstrings wird im Abschnitt »Themendateien erstellen« ausführlich beschrieben.

Titel zuweisen: Die Title-Option

Mit der Option *Title* können Sie Ihrem Hilfesystem einen Titel zuweisen. Der Titel wird in der Titelzeile des Hilfefensters angezeigt. Das Wort »Help«, gefolgt von dem Dateinamen der Hilfe-Ressourcendatei, wird dem Titel automatisch angehängt. Die Syntax für *Title* lautet:

TITLE=Hilfesystem-Titel

Titel dürfen maximal 32 Zeichen lang sein. Wenn Sie keinen Titel mit der *Title*-Option spezifizieren, wird nur das Wort »Help«, gefolgt von dem Dateinamen der Hilfe-Ressourcendatei, in der Titelzeile angezeigt. Da der Compiler das Wort »Help« immer einfügt, sollten Sie es bei der Vergabe Ihres Titels nicht verwenden.

Schriften konvertieren: Die Forcefont-Option

Mit der Option *Forcefont* können Sie eine Hilfe-Ressourcendatei erstellen, die nur eine Schriftart verwendet. Dies ist bei der Compilierung eines Hilfesystems sinnvoll, dessen Themendateien Schriften enthalten, die von den Anwendersystemen nicht unterstützt werden. Die Syntax für *Forcefont* lautet:

FORCEFONT=Schriftname

Der Parameter *Schriftname* ist eine beliebige Windows-Systemschrift. Der in der *Forcefont*-Option angegeben *Schriftname* darf keine Leerzeichen enthalten. Die Schrift *Tms Rmn* kann deshalb nicht verwendet werden.

Geben Sie die Namen der Schriften genau so ein, wie sie im Dialogfenster *Schriftarten* des Systemmenüs angezeigt werden. Schriftnamen dürfen maximal 20 Zeichen lang sein. Wenn Sie eine Schrift zuweisen, die der Compiler nicht akzeptiert, wird eine Fehlermeldung erzeugt und die Compilierung mit der Standardschrift Helvetica (Helv) fortgesetzt.

Im Abschnitt »Schriften und Schriftgrößen« weiter vorne in diesem Kapitel sind die mitgelieferten Windows-Schriften aufgelistet.

Schriftgrößen ändern: Die Mapfontsize-Option

Die Schriftgrößen in den Themendateien können mit der *Mapfontsize*-Option in andere Größen umgewandelt werden. Sie können auf diese Weise Ihren Text in einer Größe erfassen und editieren, die ein leichteres Lesen auf dem Bildschirm ermöglicht, und ihn später durch den Compiler für die tatsächliche Ausgabe im Hilfesystem umwandeln lassen. Die Syntax für *Mapfontsize* lautet:

MAPFONTSIZE=m[-n]:p

Der Parameter *m* gibt die Größe der »Quellschrift« an, während *p* die gewünschte Größe für die Ressourcendatei bezeichnet. Alle Schriften in den Themendateien mit der Größe *m* werden in die Größe *p* umgewandelt. Mit dem optionalen Parameter *n* kann ein Größenbereich für die Umwandlung spezifiziert werden. Alle Schriften in den Themendateien mit einer Größe zwischen *m* und *n* werden in die Größe *p* umgewandelt. Die folgenden Beispiele illustrieren den Gebrauch der *Mapfontsize*-Option:

```
MAPFONTSIZE=12-24:16 ;Die Größen zwischen 12 und 24 sollen in Größe
                     ;16 umgewandelt werden
MAPFONTSIZE=8:12     ;Die Größe 8 soll in Größe 12 umgewandelt werden
```

In jeder *Mapfontsize*-Anweisung kann nur eine Größe bzw. ein Größenbereich angegeben werden. Wenn Sie im [Options]-Abschnitt mehrere *Mapfontsize*-Anweisungen angeben, kann es durch verschiedene Quellgrößen bzw. Quellgrößenbereiche zu Diskrepanzen kommen. Im folgenden Beispiel ergibt sich durch die zweite Anweisung ein Fehler:

```
MAPFONTSIZE=12-24:16  MAPFONTSIZE=14:20
```

Sehen Sie sich noch einmal das erste Beispiel zur *Mapfontsize*-Option an: Die zweite Anweisung enthält eine Größe, die bereits in der ersten Anweisung umgewandelt wurde. Sie wird deshalb vom Compiler ignoriert. In einer Projektdatei können maximal fünf Größenbereiche angegeben werden.

Zusätzliche Schlüsselworttabellen: Die Multikey-Option

Mit der *Multikey*-Option können Sie ein Zeichen für eine zusätzliche Schlüsselworttabelle festlegen. Die Syntax für *Multikey* lautet:

MULTIKEY=Fußnotenzeichen

Der Parameter *Fußnotenzeichen* spezifiziert das Zeichen für die Schlüsselwortfußnote. Die Groß-/Kleinschreibung wird dabei berücksichtigt. Im folgenden Beispiel wird der Buchstabe L als Fußnotenzeichen für eine Schlüsselworttabelle vergeben:

```
MULITKEY=L
```

Hinweis: Verwenden Sie für Ihre Fußnoten nur eine Schreibweise – also entweder Groß- oder Kleinschreibung. Im obigen Beispiel werden die Schlüssel-

wörter von Themen mit der Fußnote L in eine zusätzliche Schlüsselworttabelle eingebracht, solche mit der Fußnote l dagegen nicht.

Sie können für Ihre Fußnoten jedes alphanumerische Zeichen außer »K« und »k« verwenden. Diese beiden Buchstaben sind für die normale Schlüsselworttabelle des Hilfesystems reserviert. Ein Hilfesystem kann maximal fünf Schlüsselworttabellen – die normale Tabelle inbegriffen – enthalten. Meist werden Sie aber – abhängig von der Systemkonfiguration und der Struktur Ihres Hilfesystems – nur zwei oder drei verwenden. Kann der Compiler eine zusätzliche Tabelle nicht anlegen, wird sie bei der Compilierung ignoriert.

Datei komprimieren: Die Compress-Option

Mit der Option *Compress* können Sie die Größe der vom Compiler erzeugten Hilfe-Ressourcendatei reduzieren. Der tatsächliche Umfang der Komprimierung hängt von der Anzahl, der Größe und der Komplexität der compilierten Themen ab. Im allgemeinen gilt: Je größer die Hilfedateien, desto stärker können Sie komprimiert werden. Die Syntax für *Compress* lautet:

COMPRESS=TRUE | FALSE

Da das Hilfesystem komprimierte Dateien sehr schnell laden kann, ist es vorteilhaft, komprimierte Hilfedateien zu erstellen und an die Anwender weiterzugeben. Die Dauer der Compilierung kann sich allerdings bei eingeschalteter Komprimierung erhöhen, da das Assemblieren und Sortieren einer Schlüsselsatztabelle länger dauert. Im Frühstadium des Projekts ist es daher besser, ohne Komprimierung zu compilieren.

Wenn Sie die Option *Compress* verwenden, komprimiert der Compiler das Hilfesystem, indem er Sätze, die in den Quelldateien mehrfach vorkommen, kombiniert. Der Compiler erzeugt dabei eine Satztabellendatei mit der Namenserweiterung .PH, falls diese nicht bereits vorhanden ist. Findet der Compiler eine Datei mit der Erweiterung .PH, benutzt er diese für die aktuelle Compilierung. Die Komprimierung geht in diesem Fall schneller vor, wenn seit der letzten Compilierung keine großen textuellen Änderungen vorgenommen wurden.

Sie können verhindern, daß der Compiler die vorhandene .PH-Datei verwendet, indem Sie diese vor der Compilierung löschen. Die maximale Komprimierung ist nur möglich, wenn der Compiler eine neue Satztabelle erstellen muß.

Alternative Kontextstrings angeben: Der [Alias]-Abschnitt

Im [Alias]-Abschnitt können Sie einem Themennamen (topic alias) einen oder mehrere Kontextstrings zuweisen. Ein derartiger Kontextstring kann für jedes Thema nur einmal vergeben werden und darf nicht für andere Themen im gleichen Hilfeprojekt verwendet werden. Der [Alias]-Abschnitt ermöglicht Ihnen daher das Löschen oder Kombinieren von Hilfethemen, ohne daß die Dateien selbst neu geschrieben werden müssen. Die Verwendung des [Alias]-Abschnitts ist optional.

Angenommen, Sie erstellen ein Thema, das die Informationen dreier anderer Themen ersetzt. Wenn Sie diese drei Themen löschen, müssen Sie Ihre Dateien

nach nunmehr ungültigen Kreuzreferenzen durchsuchen. Sie können sich diesen zeitraubenden Vorgang ersparen, indem Sie den drei gelöschten Themen im [Alias]-Abschnitt den Namen des neuen Themas zuweisen. Sie können den [Alias]-Abschnitt auch einsetzen, wenn Ihre Anwendung mehrere Kontextbezeichner enthält, für die Sie nur ein Thema zur Verfügung haben (das kann bei einem kontextabhängigen Hilfesystem der Fall sein).

Die Ausdrücke im [Alias]-Abschnitt müssen folgendes Format haben:

Kontext_string=Alias

Der Parameter *Alias* steht für den alternativen String (den Alias-Namen), der Parameter *kontext_string* ist der Kontextstring, der ein bestimmtes Thema kennzeichnet. Ein Alias-String hat das gleiche Format und muß dieselben Konventionen erfüllen wie der alte Kontextstring. D.h. Groß-/Kleinschreibung wird nicht berücksichtigt und die Buchstaben A bis Z, die Ziffern 0 bis 9, Punkte und Unterstriche können enthalten sein.

Das folgende Beispiel zeigt, wie ein [Alias]-Abschnitt aussehen kann:

```
[ALIAS]
sm_key=key_shrtcuts
cc_key=key_shrtcuts
st_key=key_shrtcuts ;kombiniert im Thema für Tastenkürzel
clskey=us_dlog_bxs
maakey=us_dlog_bxs  ;im Thema für Dialogfenster enthalten
chk_key_dlogprts
drp_key_dlogprts
lst_key_dlogprts
opt_key_dlogprts
tbx_key_dlogprts    ;kombiniert in Teilen des Themas für
                    ;Dialogfenster
frmtxt=edittxt
wrptxt=edittxt
seltxt=edittxt      ;im Thema für das Editieren von Text enthalten
```

Hinweis: Alias-Namen können im [Map]-Abschnitt der Hilfe-Projektdatei verwendet werden. In diesem Fall muß aber der [Alias]-Abschnitt vor dem [Map]-Abschnitt stehen.

Kontextabhängige Themen anpassen: Der [Map]-Abschnitt

Unterstützt Ihr Hilfesystem kontextabhängige Hilfe, können Sie im [Map]-Abschnitt den Kontextnummern entweder Kontextstrings oder Alias-Namen zuweisen. Die Kontextnummer entspricht einem Wert, den die übergeordnete Anwendung an das Hilfesystem übergibt, damit diese ein bestimmtes Thema anzeigt. Der Abschnitt [Map] ist optional.

Beim Schreiben des [Map]-Abschnitts haben Sie folgende Möglichkeiten:

- Zur Angabe der Kontextnummern können Sie dezimale oder hexadezimale Nummern verwenden, die dem Format der Standard-C-Notation entsprechen.

- Weisen Sie einem Kontextstring jeweils nur eine Kontextnummer oder einen Alias-Namen zu. Wenn Sie dieselbe Nummer mehreren Kontextstrings zuweisen, hat dies einen Compiler-Fehler zur Folge.
- Trennen Sie Kontextnummern und Kontextstrings durch Whitespace-Zeichen. Verwenden Sie dazu entweder Leerzeichen oder Tabs.

Mit Hilfe der **#include**-Anweisung können Sie andere Dateien in die Zuweisung einbeziehen. Der [Map]-Abschnitt unterstützt ein erweitertes Format, das die direkte Verwendung von C-Dateien mit der Namenserweiterung .H ermöglicht.

Einträge, die dieses Format verwenden, müssen mit der **#define**-Anweisung beginnen und können Kommentare im C-Format enthalten. Hierzu folgendes Beispiel:

```
#define    context_string    context_number     /* Kommentar */
```

Das folgende Beispiel zeigt verschiedene Formate, die Sie im [Map]- Abschnitt verwenden können.

```
[MAP]
```

1. Diese acht Einträge stellen jeweils die hexadezimale Entsprechung der Kontextnummer dar.

```
Edit_Window        0x0001
Control_Menu       0x0002
Maximize_Icon      0x0003
Minimize_Icon      0x0004
Split_Bar          0x0005
Scroll_Bar         0x0006
Title_Bar          0x0007
Window_Border      0x0008
```

2. Diese fünf Einträge zeigen dezimale Kontextnummern.

```
dcmb_scr           30; Document Control-menu Icon
dmxi_scr           31; Document Maximize Icon
dmni_scr           32; Document Minimize Icon
dri_scr            33; Document Restore Icon
dtb_scr            34; Document Title Bar
```

3. Diese fünf Einträge zeigen, wie Themen eingebunden werden können, die in einer C-Include-Datei definiert sind.

```
#define vscroll    0x010A /* Vertical Scroll Bar */
#define hscroll    0x010E /* Horizontal Scroll Bar */
#define scrollthm  0x0111 /* Scroll Thumb */
#define upscroll   0x0112 /* Up Scroll Arrow */
#define dnscroll   0x0113 /* Down Scroll Arrow */
```

4. Dieser Eintrag zeigt eine **#include**-Anweisung für generische Themen.

```
#include <beispiel.h>
```

Wenn in Kontextnummern die **#define**-Anweisung verwendet wird und die Datei mit den **#define**-Anweisungen im Quelltext der Anwendung *und* in der Hilfedatei enthalten ist, werden Aktualisierungen der Kontextnummern durch den Anwendungsprogrammierer automatisch in der nächsten Compilierung berücksichtigt.

Sie können die im [Map]-Abschnitt aufgelisteten Kontextstrings entweder in einem Hilfethema oder im [Alias]-Abschnitt definieren. Der Compiler erzeugt eine Fehlermeldung, wenn ein Kontextstring im [Map]-Abschnitt, nicht jedoch in einer der Themendateien oder im [Alias]-Abschnitt enthalten ist.

Hinweis: Wenn Sie einen Alias-Namen verwenden, muß in der Hilfe-Projektdatei der [Alias]-Abschnitt vor dem [Map]-Abschnitt stehen.

Der Abschnitt »Kontextabhängige Hilfethemen anzeigen« im Abschnitt »Das Hilfesystem planen« enthält weitere Informationen zur kontextabhängigen Hilfe.

Referenzierte Bitmaps: Der [Bitmaps]-Abschnitt

Wenn Sie in Ihrem Hilfesystem referenzierte Bitmaps verwenden, müssen die Namen der Dateien in dem [Bitmaps]-Abschnitt aufgelistet werden. Im folgenden Beispiel sehen Sie das Format dieses [Bitmaps]-Abschnitts:

```
[BITMAPS]
DUMP01.BMP
DUMP02.BMP
DUMP03.BMP
c:\PROJECT\HELP\BITMAPS\DUMP04.BMP
```

Hinweis: Für den [Bitmaps]-Abschnitt gelten dieselben Regeln wie für den [Files]-Abschnitt.

Hilfedateien compilieren

Sobald die Projektdatei angelegt ist, können Sie mit dem Hilfe-Compiler eine Hilfedatei compilieren.

Der Compiler erzeugt aus den in der Projektdatei aufgeführten Themendateien die binäre Ressourcendatei. Nach diesem Prozeß kann die Anwendung auf die daraus resultierende Ressourcendatei zugreifen.

Bedenken Sie vor der Compilierung die Position folgender Dateien:

- Der Hilfe-Compiler HC.EXE muß sich in einem Verzeichnis befinden, von dem aus er aufgerufen werden kann, d.h entweder im aktuellen Verzeichnis oder in einem mit dem PATH-Befehl angemeldeten Verzeichnis. Sie können beim Aufruf natürlich auch einen vollständigen Pfad angeben, zum Beispiel:

```
C:\BIN\HC HELPEX.HPJ
```

- Die Projektdatei *Dateiname*.HPJ kann sich entweder im aktuellen Verzeichnis befinden, oder muß mit Angabe des vollständigen Pfads an HC.EXE übergeben werden:

```
C:\BIN\HC D:\MYPROJ\HELPEX.HPJ
```

- Die in der Projektdatei aufgeführten Themendateien, die als RTF-Dateien gespeichert sind, können sich im aktuellen Verzeichnis, in einem im [Files]-Abschnitt angegebenen Unterverzeichnis des akteullen Verzeichnisses oder in dem bei der *Root*-Option spezifizierten Verzeichnis befinden.

- Dateien, die samt ihrem Pfadnamen in der **#include**-Anweisung angegeben werden, befinden sich an den angegebenen Positionen.

- Die Positionen der in den Themendateien referenzierten Bitmap-Dateien müssen korrekt sein.

Die in einer **#include**-Anweisung angegebenen Dateien müssen sich im Suchpfad des Projekt-Verzeichnisses befinden oder mit der *Root*-Option angegeben werden. Der Compiler sucht nur nach den Verzeichnissen, die der Projektdatei festgelegt sind.

Informationen zu der Option *Root* finden Sie im Abschnitt »Arbeitsverzeichnis angeben: Die Root-Option« weiter vorne in diesem Kapitel.

Hinweis: Wenn Sie ein RAM-Laufwerk für temporäre Dateien (definiert mit der DOS-Umgebungs-Variable *TMP*) verwenden, muß seine Kapazität groß genug für die compilierte Ressourcendatei sein. Wenn die Hilfedatei größer als die verfügbare Kapazität des RAM-Laufwerks ist, erzeugt der Compiler einen Fehler und bricht die Compilierung ab.

Der Hilfe-Compiler

Für den Start des Hilfe-Compilers geben Sie den Befehl `HC` ein. HC hat keine Optionen. Sämtliche für die Compilierung benötigten Informationen müssen in der Projektdatei enthalten sein. `HC` wird mit folgender Syntax aufgerufen:

HC Dateiname.HPJ

Während der Compilierung werden Punkte auf dem Bildschirm ausgegeben, die den ablaufenden Prozeß kennzeichnen. Tritt eine Fehlerbedingung auf, erhalten Sie eine entsprechende Meldung.

Wenn der Hilfe-Compiler die Compilierung beendet hat, schreibt er eine Ressourcendatei mit der Dateinamenserweiterung .HLP in das aktuelle Verzeichnis. Danach erscheint die DOS-Eingabeaufforderung.

Die bei der Compilierung erstellte Ressourcendatei hat denselben Namen wie die Projektdatei.

Compiler-Fehlermeldungen und Statusmeldungen können mit der üblichen DOS-Syntax in eine Datei umgeleitet werden. Das ist bei umfangreichen Compilierungen sinnvoll, die nicht insgesamt aufgezeichnet werden sollen. Die umgeleitete Datei wird als Textdatei gespeichert und kann jederzeit in einen beliebigen ASCII-Editor geladen werden.

Schnittstelle zum Hilfesystem programmieren

Das Autorenteam muß die Anwendung so programmieren, daß der Anwender über das Windows-Hilfesystem auf die Hilfedatei zugreifen kann. Das Hilfesystem ist eine selbständige Windows-Anwendung. Ihre eigene Anwendung kann Windows dazu veranlassen, das Hilfesystem zu starten und das Thema festzulegen, das im Hilfefenster angezeigt werden soll. Für den Anwender stellt sich das Windows-Hilfesystem somit als ein Teil der Anwendung dar.

WinHelp von einer Anwendung aus aufrufen

Eine Anwendung stellt dem Anwender ein Hilfesystem durch den Aufruf der Funktion *WinHelp* zur Verfügung. *WinHelp* hat folgende Syntax:

BOOL WinHelp (hWnd, lpHelpFile, wCommand, dwData)

Entsprechende C- und Pascal-Beispiele finden Sie auf Ihren Originaldisketten.

Der Parameter *hWnd* kennzeichnet das Fenster, das Hilfe anfordert. Das Windows-Hilfesystem verwendet diesen Bezeichner zur Überwachung der Anwendungen, die Hilfe angefordert haben.

Der Parameter *lpHelpFile* gibt den Namen (optional mit Pfadangabe) der Hilfedatei an, die das gewünschte Thema enthält.

Der Parameter *wCommand* legt zweierlei fest: Entweder den Typ der Suche, die das Hilfesystem für das Auffinden eines speziellen Themas verwendet, oder daß das Hilfesystem von der Anwendung nicht länger benötigt wird. Der Parameter kann mit einem der folgenden Werte besetzt werden:

Tabelle A.12 Die Werte von wCommand

Wert	Bedeutung
HELP_CONTEXT	Zeigt Hilfeinformationen zu einem bestimmten Thema an.
HELP-HELPONHELP	Zeigt das Thema zur Verwendung des Hilfeindex an.
HELP_INDEX	Zeigt den Hauptindex des Hilfesystems an.
HELP_KEY	Zeigt Informationen zu einem Thema an, das durch ein Schlüsselwort gekennzeichnet ist.
HELP-MULTIKEY	Zeigt Information zu einem Thema an, das von einem Schlüsselwort in einer zusätzlichen Schlüsselworttabelle gekennzeichnet ist.
HELP-QUIT	Informiert das Hilfesystem darüber, daß Hilfe nicht länger benötigt wird. Fordert keine andere Anwendung das Hilfesystem an, schließt Windows das Hilfesystem.
HELP_SETINDEX	Zeigt ein bestimmtes Thema des Hilfeindex an.

Der Parameter *dwData* legt das Thema fest, für das die Anwendung das Hilfesystem aufruft. Das Format von *dwData* hängt von dem Wert ab, der von *wCommand* beim Aufruf von *WinHelp* übergeben wird.

In der folgenden Tabelle finden Sie das Format von *dwData* für jeden Wert von *wCommand*:

Tabelle A.13 Die Formate von dwData

wCommand-Wert	*dwData*-Format
HELP_CONTEXT	Ein vorzeichenloser Integerwert vom Typ **long**, der die Kontextnummer des Themas enthält. Anstelle von HELP_-INDEX kann HELP_CONTEXT den Wert -1 verwenden
HELP_HELPONHELP	Wird ignoriert
HELP_INDEX	Wird ignoriert.
HELP_KEY	Ein Zeiger vom Typ **long** auf einen String, der ein Schlüsselwort für das gewünschte Thema enthält.
HELP_MULTIKEY	Ein Zeiger vom Typ **long** auf eine MULTIKEY-Struktur, wie sie in WINDOWS.H definiert ist. Diese Struktur legt das Tabellen-Fußnotenzeichen und das Schlüsselwort fest.
HELP_QUIT	Wird ignoriert.
HELP_SETINDEX	Ein vorzeichenloser Integerwert vom Typ **long**, der die Kontextnummer des Themas enthält.

Da die Funktion *WinHelp* sowohl eine Kontextnummer als auch ein Schlüsselwort angegeben kann, unterstützt sie die kontexabhängige und Themenbezogene Suche in der Hilfedatei.

Hinweis: Um sicherzustellen, daß der Index korrekt gesetzt bleibt, sollte in Ihrer Anwendung jedem Aufruf von *WinHelp*, bei dem *wCommand* auf HELP_CONTEXT gesetzt wurde, ein weiterer Aufruf von *WinHelp* folgen, bei dem *wCommand* auf HELP_SETINDEX (mit *dwData* wird der entsprechende Kontextbezeichner angegeben) gesetzt ist. HELP_INDEX sollte nie zusammen mit HELP_SETINDEX verwendet werden.

Kontextabhängige Hilfe

Kontextabhängige Hilfe sollte dann zur Verfügung gestellt werden, wenn der Anwender etwas über den Zweck eines bestimmten Fensters oder Befehls erfahren möchte. So könnte der Anwender beispielsweise das Dateimenü öffnen, den Befehl *Öffnen* anwählen (mit Hilfe der Pfeiltasten) und dann F1 drücken, um Informationen zu diesem Punkt zu erhalten.

Die Implementierung kontextabhängiger Hilfe erfordert fortgeschrittene Programmiertechniken. Die Beispielanwendung *Helpex* verdeutlicht den Einsatz zweier Techniken, die in den folgenden Abschnitten diskutiert werden.

Unterstützung von Umschalt F1 Um den Umschalt F1 -Modus zu implementieren, muß *Helpex* auf diese Akzelerator-Taste mit dem Aufruf von *Set-Cursor* antworten, um die Darstellungsform des Mauszeigers zu ändern. Dieser nimmt dann die Gestalt eines Pfeils ergänzt durch ein Fragezeichen an.

```
case WM_KEYDOWN:
     if (wParam == VK_F1)   {

         /* Bei Umschalt-F1 Einschalten des Hilfemodus und Setzen
            des Hilfezeigers */

     if (GetKeyState(VK_SHIFT))   {
        bHelp = TRUE;
        SetCursor(hHelpCursor);
        return (DefWindowProc(hWnd, message,
               wParam, lParam));
     }

     /* Bei F1 ohne Umschalt Aufruf des Hilfehauptindex-Themas */
     else   {
        WinHelp(hWnd,szHelpFileName,HELP_INDEX,0L);
     }
}

else if (wParam == VK_ESCAPE && bHelp)   {

     /* ESC während des Hilfemodus: Ausschalten des
        Hilfemodus */
     bHelp = FALSE;
     SetCursor ((HCURSOR)GetClassWord(hWnd,GCW_HCURSOR));
}
break;
```

Solange sich der Anwender im Hilfemodus befindet (das heißt, bis er mit der Maus klickt oder Esc drückt), antwortet *Helpex* auf WM_SELECTCURSOR-Botschaften mit dem erneuten Setzen des Mauszeigers auf die Kombination von Pfeil und Fragezeichen.

```
case WM_SETCURSOR:
  /* Im Hilfemodus muß der Mauszeiger als Antwort auf jede
     wm_SetCursor-Botschaft neu gesetzt werden. Andernfalls
     setzt Windows per Voreinstellung den Mauszeiger auf den
     der Fensterklasse entsprechenden Zeiger. */
   if (bHelp)  {
      SetCursor(hHelpCursor);
      break;
   }
   return (DefWindowProc(hWnd, message, wParam, lParam);
   break;
case MW_INITMENU:
   if (bHelp)  {
       SetCursor(hHelpCursor);
   }
   return  (TRUE);
```

Wenn sich der Anwender im Umschalt F1 -Modus befindet und den Mausknopf drückt, empfängt *Helpex* eine WM_NCLBUTTONDOWN-Meldung, wenn

der Klick in einem Nicht-Client-Bereich des Anwendungsfensters ausgeführt wurde.

Durch Überprüfen des *wParam*-Wertes dieser Meldung kann bestimmt werden, welche Kontext-ID an *WinHelp* übergeben wird.

```
case WM_NCLBUTTONDOWN:
   /* Im Hilfemodus (Umschalt+F1) wird kontextabhängige Hilfe für
      den Nicht-Client-Bereich angezeigt. */
   if (bHelp)  {
       dwHelpContextID =
           (wParam == HTCAPTION) ? (DWORD)HELPID_TITLE_BAR:
           (wParam == HTSIZE) ? (DWORD)HELP_ID_SIZE_BOX:
           (wParam == HTREDUCE) ? (DWORD)HELPID_MINIMIZE_ICON:
           (wParam == HTZOOM) ? (DWORD)HELPID_MAXIMIZE_ICON:
           (wParam == HTSYSMENU) ? (DWORD) HELPID_SYSTEM_MENU:
           (wParam == HTBOTTOM) ? (DWORD)HELPID_SIZING_BORDER:
           (wParam == HTBOTTOMLEFT) ? (DWORD)HELPID_SIZING_BORDER:
           (wParam == HTBOTTOMRIGHT) ? (DWORD)HELPID_SIZING_BORDER:
           (wParam == HTTOP) ? (DWORD)HELPID_SIZING_BORDER:
           (wParam == HTLEFT) ? (DWORD)HELPID_SIZING_BORDER:
           (wParam == HTRIGHT) ? (DWORD)HELPID_SIZING_BORDER:
           (wParam == HTTOPLEFT) ? (DWORD) HELPID_SIZING_BORDER:
           (wParam == HTTOPRIGHT) ? (DWORD)HELPID_SIZING_BORDER:
           (DWORD) 0L;
       if (! ((BOOL)dwHelpContextId))
           return (DefWindowProc(hWnd, message, wParam, lParam));
       bHelp = FALSE;
     WinHelp(hWnd,szHelpFileName,HELP_CONTEXT,dwHelpContextID);
       break;
   }

   return (DefWindowProc(hWnd, message, wParam, lParam);
```

Unterstützung von F1 Die kontextabhängige Unterstützung von F1 für Menüs ist relativ einfach in Ihrer Anwendung zu implementieren. Wenn ein Menü geöffnet ist und der Anwender F1 drückt, während ein Menüauswahlpunkt hervorgehoben ist, sendet Windows eine WM_ENTERIDLE-Botschaft an die Anwendung. Damit wird gekennzeichnet, daß das System in den *idle*-Status zurückkehrt, wenn festgestellt wurde, daß F1 kein gültiger Tastendruck für die Auswahl eines Menüpunktes ist. Sie können den *idle*-Status dafür verwenden, um sich den Tastatur-Status zum Zeitpunkt der WM_ENTERIDLE-Botschaft anzusehen.

Wenn F1 gedrückt ist, können Sie das Betätigen der Taste CR durch den Anwender simulieren, indem Sie eine WM_KEYDOWN-Botschaft mit VK_-RETURN abschicken. Da Sie aber nicht wollen, daß die Anwendung den Menübefehl wirklich ausführt, müssen Sie das Flag (bHelp=TRUE) setzen. Dies bewirkt, daß Ihre Anwendung, wenn sie die Botschaft WM_COMMAND für den Menüpunkt erhält, den Befehl nicht ausführt. Stattdessen zeigt das Hilfesystem das Hilfethema für diesen Menüpunkt an.

Die folgenden Beispiele zeigen den Einsatz von F1 für Menüauswahlpunkte.

```
case WM_ENTERIDLE:
    if ((wParam == MSGF_MENU) && (GetKeyState(VK_F1) & 0x8000))  {
        bHelp = TRUE;
        PostMessage(hWnd, WM_KEYDOWN, VK_RETURN, 0L);
    }
    break;

case WM_COMMAND:
    /* Wurde gerade F1 gedrückt oder ist der
       Hilfemodus (Umschalt+F1) aktiv? */

    if (bHelp)  {
       dwHelpContextId =
           (wParam == IDM_NEW) ? (DWORD)HELPID_FILE_NEW:
           (wParam == IDM_OPEN) ? (DWORD)HELPID_FILE_OPEN:
           (wParam == IDM_SAVE) ? (DWORD)HELPID_FILE_SAVE:
           (wParam == IDM_SAVEAS) ? (DWORD)HELPID_FILE_SAVE_AS:
           (wParam == IDM_PRINT) ? (DWORD)HELPID_FILE_PRINT:
           (wParam == IDM_EXIT) ? (DWORD)HELPID_FILE_EXIT:
           (wParam == IDM_UNDO) ? (DWORD)HELPID_EDIT_UNDO:
           (wParam == IDM_CUT) ? (DWORD)HELPID_EDIT_CUT:
           (wParam == IDM_CLEAR) ? (DWORD)HELPID_EDIT_CLEAR:
           (wParam == IDM_COPY) ? (DWORD)HELPID_EDIT_COPY:
           (wParam == IDM_PASTE) ? (DWORD)HELPID_EDIT_PASTE:
           (DWORD) 0L;

       if (!dwHelpContextId)
       {MessageBox(hWnd,
        "Keine Hilfe für den Hilfemenüeintrag vorhanden",
        "Hilfe-Beispiel" MB_OK
        return (DefWindowProc(hWnd, message, wParam, lParam));
       }
      bHelp = FALSE;
    WinHelp(hWnd,szHelpFileName,HELP_CONTEXT,dwHelpContextId);
      break;
   }
```

In Dialogfenstern ist es schwieriger als in Menüs festzustellen, daß F1 gedrückt wurde. Sie müssen einen Botschaftsfilter installieren und zwar mit der WH_MSGFILTER-Option der Funktion SetWindowsHook. Die Botschaftsfilter-Funktion antwortet auf WM_KEYDOWN- und WM_KEYUP-Botschaften für VK_F1, wenn diese durch einen MSGF_DIALOGBOX-Code an ein Dialogfenster geschickt werden. Durch Überprüfung der an den Filter übergebenen Botschaftsstruktur können Sie den Kontext von F1 bestimmen: das Dialogfenster und die Option oder den Eintrag.

Während der Bearbeitung der gefilterten Botschaften sollten Sie nicht *WinHelp* aufrufen, sondern eine anwendungsdefinierte Botschaft an Ihre Anwendung schicken, die *WinHelp* bei der ersten sich bietenden Möglichkeit aufrufen soll.

Hilfe für einen im Hilfemenü aufgeführten Eintrag

Es gibt Situationen, in denen der Anwender Hilfeinformationen über ein generelles Konzept der Anwendung haben möchte und nicht nur über ein bestimmtes Steuerelement oder Fenster. In diesem Fall sollte die Anwendung Hilfe für ein bestimmtes Thema über ein Schlüsselwort und nicht über einen Kontextbezeichner zur Verfügung stellen.

Wenn die Hilfedatei Ihrer Anwendung beispielsweise ein Thema enthält, das den Gebrauch der Tastatur beschreibt, sollten Sie einen Eintrag mit dem Wortlaut »Tastatur« in Ihrem Hilfemenü haben. Wenn der Anwender diesen Begriff anwählt, ruft Ihre Anwendung *WinHelp* auf und fordert das entsprechende Thema an:

```
case IDM_HELP_KEYBOARD:
   WinHelp (hWnd, lpHelpFile, Help_KEY, (LPSTR) "Tastatur");
   break;
```

Auf zusätzliche Schlüsselworttabellen zugreifen

In Ihrer Anwendung können Befehle oder Begriffe eingesetzt sein, die Begriffen in ähnlichen anderen Anwendungen entsprechen. Wenn ein Schlüsselwort angegeben wird, kann Ihre Anwendung *WinHelp* aufrufen und Themen nachschlagen, die in einer zusätzlichen Schlüsselworttabelle stehen. Diese *Multikey*-Funktion wird über *WinHelp* erreicht, indem der Parameter *wCommand* auf HELP_MULTIKEY gesetzt wird.

Sie geben über eine *Multikeyhelp*-Struktur ein Fußnotenzeichen für die zusätzliche Schlüsselworttabelle und das Schlüsselwort oder den Satz ein. Die Struktur wird als *dwData*-Parameter in den Aufruf von *WinHelp* übergeben. Diese Struktur ist in windows.h folgendermaßen definiert:

```
typedef struct tagMULTIKEYHELP  {
   WORD mdSize;
   BYTE mkKeyList;
   BYTE szKeyPhrase[1];
} MULTIKEYHELP;
```

Die folgende Tabelle enthält das Format der *Multikeyhelp*-Strukturfelder:

Tabelle A.14 Formate der Multikeyhelp-Strukturfelder

Parameter	Format
mkSize	Die Größe der Struktur, einschließlich des Schlüsselworts (oder Satzes) und des damit verknüpften Kennbuchstabens.
mkKeyList	Ein Einzelzeichen, das das Fußnotenzeichen für die zusätzliche Schlüsselworttabelle angibt, die gesucht werden soll.
szKeyPhrase	Ein nullterminiertes Schlüsselwort (bzw. Satz) nach dem in der zusätzlichen Schlüsselworttabelle gesucht werden soll.

Das folgende Beispiel zeigt eine Schlüsselwortsuche für das Wort »Rahmen« in der durch das Fußnotenzeichen »L« gekennzeichneten zusätzlichen Schlüsselworttabelle:

```
MULTIKEYHELP mk;
char szKeyword[]="Rahmen";
mk.mkSize=sizeof(MULTIKEYHELP) + (WORD)lstrlen(szKeyword);
mk.mkKeyList="L";
mk.szKeyphrase=szKeyword;
WinHelp(hWnd,lpHelpfile,HELP_MULTIKEY,(LPSTR)&mk);
```

Hilfe beenden

Das Windows-Hilfesystem ist eine Ressource, die alle Windows-Anwendungen benutzen können. Da sie außerdem auch eine selbständige Anwendung ist, kann der Anwender sie wie jede andere Anwendung ausführen. Als Folge davon hat Ihre Anwendung nur eine beschränkte Kontrolle über das Hilfesystem. Ihre Anwendung kann das Hilfefenster nicht direkt schließen, allerdings das Hilfesystem davon informieren, daß es nicht länger benötigt wird. Ihre Anwendung sollte vor dem Schließen des Hauptfensters *WinHelp* aufrufen und den *wCommand*-Parameter auf HELP_QUIT setzen. Dadurch wird das Hilfesystem davon in Kenntnis gesetzt, daß es nicht länger von Ihrer Anwendung benötigt wird. Sehen Sie sich dazu das folgende Beispiel an:

```
case WM_DESTROY:
   WinHelp (hWnd, lpHelpFile, HELP_QUIT, NULL);
```

Eine Anwendung, die *WinHelp* während ihrer Ausführung aufgerufen hat, muß *WinHelp* mit dem *wCommand*-Parameter HELP_QUIT aufrufen, bevor sie *WinMain* verläßt (normalerweise während der Bearbeitung der WM_DESTROY-Botschaft). Wenn eine Anwendung mehrere Hilfedateien öffnet, muß für die Beendigung jeder Datei *WinHelp* aufgerufen werden.

Wenn eine Anwendung oder eine dynamische Linkbibliothek eine Hilfedatei geöffnet hat, aber die verknüpfte Instanz der Hilfe-Engine (WINHELP.EXE) nicht länger aktiv bleiben soll, muß die Anwendung bzw. die DLL *WinHelp* mit dem *wCommand*-Parameter HELP_QUIT aufrufen, um die Instanz der Hilfe-Engine zu zerstören.

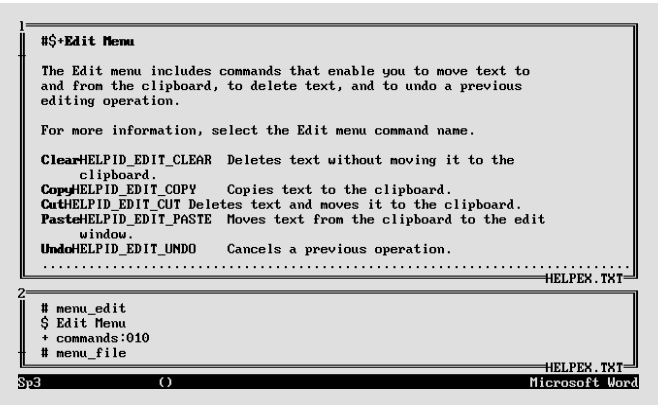
Die Anwendung oder DLL sollte auf keinen Fall beendet werden, ohne daß vorher *WinHelp* für jede der geöffneten Hilfedateien aufgerufen wurde. Eine Hilfedatei gilt als geöffnet, wenn ein vorheriger *WinHelp*-Aufruf unter Angabe des Hilfedateinamens stattgefunden hat.

Das Windows-Hilfesystem ist erst beendet, wenn alle Fenster, die *WinHelp* aufgerufen haben, *WinHelp* und den *wCommand*-Parameter HELP_QUIT aufgerufen haben. Führt eine Anwendung dies nicht aus, bleibt das Hilfesystem aktiv, auch wenn alle Anwendungen, die auf sie zugreifen, bereits geschlossen sind.

Hilfe-Beispiele

In diesem Abschnitt finden Sie einige Beispiele für Hilfe-Quelldateien und die entsprechende Anzeige der Themen im Hilfesystem. Jedes Beispiel zeigt ein Thema (oder einen Teil eines Themas), wie es der Autor mit einem RTF-fähigen

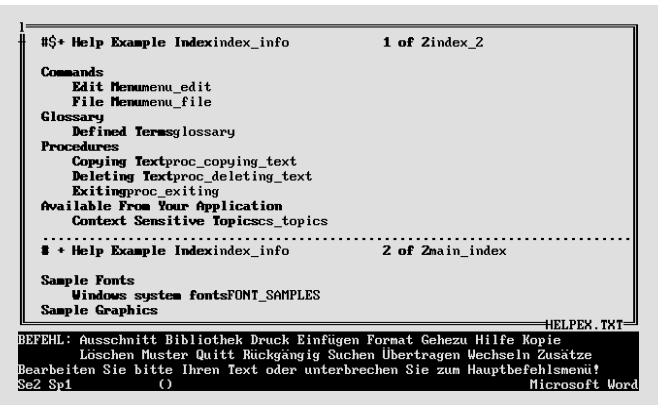
Textprogramm erfaßt und wie es der Anwender dann im Hilfefenster sieht. Sie können diese Beispiele als Ausgangspunkt für die Erstellung Ihrer eigenen Themendateien heranziehen. Die Beispiele sollen Ihnen einen Eindruck davon vermitteln, wie sich die mit einem Textprogramm erfaßten Themen dem Anwender im Hilfefenster darstellen.



```
1
#$+Edit Menu

The Edit menu includes commands that enable you to move text to
and from the clipboard, to delete text, and to undo a previous
editing operation.

For more information, select the Edit menu command name.

ClearHELPID_EDIT_CLEAR  Deletes text without moving it to the
     clipboard.
CopyHELPID_EDIT_COPY    Copies text to the clipboard.
CutHELPID_EDIT_CUT Deletes text and moves it to the clipboard.
PasteHELPID_EDIT_PASTE  Moves text from the clipboard to the edit
     window.
UndoHELPID_EDIT_UNDO    Cancels a previous operation.
.....................................................................
                                                        HELPEX.TXT
2
# menu_edit
$ Edit Menu
+ commands:010
# menu_file
                                                        HELPEX.TXT
Sp3                    ()                          Microsoft Word
```

Abbildung A.8 Thema-Erfassung mit Word

Abbildung A.9 Anzeige des Hilfethemas



```
1
#$* Help Example Indexindex_info            1 of 2index_2

Commands
     Edit Menumenu_edit
     File Menumenu_file
Glossary
     Defined Termsglossary
Procedures
     Copying Textproc_copying_text
     Deleting Textproc_deleting_text
     Exitingproc_exiting
Available From Your Application
     Context Sensitive Topicscs_topics
.....................................................................
# * Help Example Indexindex_info            2 of 2main_index

Sample Fonts
     Windows system fontsFONT_SAMPLES
Sample Graphics
                                                        HELPEX.TXT
BEFEHL: Ausschnitt Bibliothek Druck Einfügen Format Gehezu Hilfe Kopie
        Löschen Muster Quitt Rückgängig Suchen Übertragen Wechseln Zusätze
Bearbeiten Sie bitte Ihren Text oder unterbrechen Sie zum Hauptbefehlsmenü!
Se2 Sp1          ()                                 Microsoft Word
```

Abbildung A.10 Thema-Erfassung mit Word 5.0 für DOS

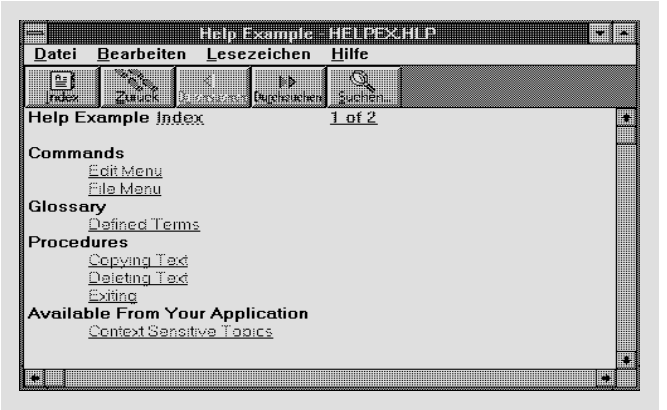


Abbildung A.11 Anzeige des Hilfethemas

Die Projektdatei für Helpex

Die Projektdatei für *Helpex* (die Beispielanwendung) sieht folgendermaßen aus:

```
[Options]
ROOT=c:\help
INDEX=main_index
TITLE=Help Example
COMPRESS=true

[Files]
helpex.rtf    ; Sprungthemen
terms.rtf     ; Definitionsbegriff

[MAP]
main_index 0xFFFF
#define HELPID_EDIT_CLEAR      100
#define HELPID_EDIT_COPY       101
#define HELPID_EDIT_CUT        102
#define HELPID_EDIT_PASTE      103
#define HELPID_EDIT_UNDO       104
#define HELPID_FILE_EXIT       200
#define HELPID_FILE_NEW        201
#define HELPID_FILE_OPEN       202
#define HELPID_FILE_PRINT      203
#define HELPID_FILE_SAVE       204
#define HELPID_FILE_SAVE_AS    205
#define HELPID_EDIT_WINDOW     300
#define HELPID_MAXIMIZE_ICON   301
#define HELPID_MINIMIZE_ICON   302
#define HELPID_SPLIT_BAR       303
```

```
#define HELPID_SIZE_BOX        304
#define HELPID_SYSTEM_MENU     305
#define HELPID_TITLE_BAR       306
#define HELPID_SIZING_BORDER   307
```

B. Turbo Editor Macro Language (TEML)

TEMC.EXE ist ein Compiler für Editormakros in der integrierten Entwicklungsumgebung (IDE). TEMC erstellt eine Makrodatei, die Editormakros und Tastenzuweisungen definiert. Außerdem wird eine Konfigurationsdatei erstellt. Diese wird von der IDE gelesen, um die Auswirkungen von Tastaturbefehlen im Editor festzulegen.

Die in die IDE eingebauten Makrodefinitionen und Tastenzuweisungen sind in der Datei DEFAULTS.TEM enthalten. Sie dient als Beispielmakro und als Grundlage der Editoranpassung.

TEMC-Kommandozeile

TEMC wird von der DOS-Kommandozeile aus gestartet. Geben Sie den Befehl

```
temc [-c] [-u] <Makrodatei> <Konfigurationsdatei>
```

ein. Die Erweiterung der Makrodatei ist .TEM, wenn Sie nichts anderes vorgeben. Für die Konfigurationsdatei geht TEMC von der Erweiterung .TC aus.

Die Konfigurationsdatei muß nicht existieren, sie wird dann aber erzeugt. Der optionale Parameter *-c* kann auch als */c* angegeben werden. Er kann an jeder Position in der Kommandozeile verwendet werden. Benutzen Sie diesen Parameter, so werden existierende Kommandotabellen in der Konfigurationsdatei verworfen, bevor die Makrodatei bearbeitet wird. Wird der Parameter *-c* nicht verwendet, verbindet TEMC die Tastenzuweisungen aus Konfigurations- und Makrodatei.

TEMC wechselt den in der IDE verwendeten Befehlssatz, wenn der alternative Befehlssatz in *Optionen/Oberfläche/Vorgaben* festgelegt ist. Der optionale Schalter *-u* (oder */u*) veranlaßt TEMC, anstelle des alternativen den CUA-Befehlssatz zu aktivieren.

Sie können DEFAULTS.TEM zur Wiederherstellung der Vorgaben verwenden. Die Datei ist als Beispiel einer Makrodatei und einer Befehlstabelle vorgesehen. Sie können sie kopieren und Ihren Wünschen anpassen. Mit Turbo C++ wird eine Datei mit dem Namen CMACROS.TEM mitgeliefert. Diese Datei enthält viele nützliche Verbesserungen der IDE für die Programmierung mit C oder C++ (je nachdem. welche Version Sie installieren).

Syntax

Die Syntax zur Definition eines Makros sieht so aus:

```
MACRO <Makroname>
  <Befehl1>;
  [ <Befehl2>; ... ]
END;
```

<Makroname> kann jedes gültige C-Symbol sein, *<Befehl>* der Name eines vordefinierten Makros oder eines vordefinierten TEMC-Editorbefehls. Eine Tabelle der Befehle und ihrer Auswirkungen findet sich am Ende dieses Anhangs.

Bei der Definition eines Makros sind folgende Punkte zu beachten:

1. Eine Anweisung definiert ein benanntes Makro oder eine Tastenzuweisung.
2. Leerstellen und Zeilenschaltungen können optional verwendet werden.
3. Kommentare werden (wie in C) in die Symbole /* ... */ eingeschlossen.
4. Im Unterschied zu C berücksichtigt TEMC Groß- und Kleinschreibung nicht.
5. Einige Editorbefehle haben eine Syntax, die einem Funktionsaufruf unter C mit einem Argument gleichen:

```
SetMark(5);
```

In Abhängigkeit vom Befehl ist das Argument entweder eine Dezimal-Integerkonstante, eine Zeichenkonstante oder ein literaler String. Sie werden unter Verwendung der C-Syntax festgelegt.

Hier das Beispiel einer Makrodefinition (aus DEFAULTS.TEM):

```
MACRO MacScrollUp
      ScrollScreenUp; FixCursorPos;
END;
```

Die Syntax der Tastaturzuweisungen lautet

```
<Tastenfolge>: <Befehl>;
```

oder

```
<Tastenfolge>: BEGIN <Befehl1>; [ <Befehl2>; ... ] END;
```

*<Tastenfolge>*kann entweder eine Zeichentaste sein, die zusammen mit Strg oder Alt gedrückt wird, oder eine Folge von Tasten, die durch ein Plus (+) getrennt werden. Die Festlegung der Zeichentasten selbst muß unter Berücksichtigung der Klein- bzw. Großschreibung vorgenommen werden. Strg-k+B ist eine andere Tastenfolge als Strg-k+b, letztere jedoch identisch mit STRG-K+b.

Whitespaces zwischen Doppelpunkt und *<Tastenfolge>* sind erlaubt. Jeder *<Befehl>* kann entweder der Name eines bereits definierten Makros oder eines vordefinierten Editorbefehls (Tabelle B.1) sein.

Tasten-schlüssel

Der Editor der IDE verwendet einen erweiterten Zeichensatz, der Tastenkombinationen beinhaltet, die normalerweise für DOS-Programme nicht zur Verfügung stehen. Tastenkürzel können in einem Makro mit einer Kombination der Symbole Strg -, Umschalt - und Alt - gefolgt von einem Zeichen festgelegt werden. Einige Tasten können nicht direkt in eine TEMC-Makrodatei eingegeben werden. Auf diese Tasten kann aber über ihre Namen nach Maßgabe der folgenden Tabelle zugegriffen werden.

Jeder Taste in einer Tastenfolge (außer der ersten) kann ein ^ oder ein @ vorangestellt werden. Das Zirkumflex (^) zeigt an, daß beim Drücken der Taste jede beliebige Kombination von Umschalt und Strg verwendet werden kann. Das Zeichen @ besagt, daß die Groß-/Kleinschreibung für das folgende Zeichen keine Rolle spielt, wobei Strg nicht akzeptiert wird. Einige Beispiele dazu:

Strg k b	legt ein Strg K , gefolgt von einem kleinen b, fest.
Strg k ^b	legt ein Strg K , gefolgt von b, B oder Strg B , fest.
Strg k @b	legt ein Strg K , gefolgt von b oder B, fest.

Tasten-benennungen

Tasten werden durch Buchstaben, Ziffern oder Zeichen festgelegt. Ihnen kann ein Strg , Alt oder Umschalt vorangehen. Folgende Namen bestimmen Tasten, die in der TEMC-Syntax nicht als sie selbst eingegeben werden können.

Tastenname	Anmerkungen
Pos1 , Ende	
Bild⇑ , Bild⇓	
Strg → , Strg ←	
⇑ , ⇓	
Einfg , Entf	
CR	
Return	Wie CR
Rock	
Tab	
Esc	
Star, Minus, Plus	Star, Plus und Minus enstprechen den Tasten *, + und CR im numerischen Tastenfeld.
Leer	
Druck	
F1 bis F10	Funktionstasten
Strg Alt F1 bis Strg Alt F10	

Vordefinierte Editorbefehle

TEMC erlaubt die Verwendung eingebauter Editorbefehle und benutzerdefinierter Makros auch als Befehle innerhalb von Makros. Dabei dürfen allerdings keine Schleifen erzeugt werden, in denen sich zwei Makros gegenseitig aufrufen. Beachten Sie, daß einige Befehle aus dem Editor aussteigen und in die IDE zurückspringen. Es gibt dann keine Möglichkeit, aus dem Makro heraus wieder in den Editor zu gelangen. Ein solcher Befehl muß also der letzte innerhalb eines Makros sein. Wird diese Regel bei der Makrodefinition nicht beachtet, erhalten Sie eine Fehlermeldung. Es folgt eine Liste aller vordefinierten TEMC-Editorbefehle. Danach folgen Befehle, die aus dem Editor aussteigen.

Tabelle B.1 TEMC-Editorbefehle

Befehlsname	Aktion des Editors
BackspaceDelete	Löscht das Zeichen links vom Cursor.
BottomOfScreen	Bewegt den Cursor in die letzte Zeile des aktuellen Fensters. Die Spaltenposition bleibt unverändert.
CenterFixScreenPos	Paßt die Bildschirmanzeige so an, daß der Cursor sichtbar wird. Benutzen Sie diese Anpassung, wenn die Cursorposition nicht im sichtbaren Bereich liegen könnte.
CopyBlock	Existiert ein gültiger markierter Textblock, wird dieser an die aktuelle Cursorposition kopiert.
CursorCharLeft	Bewegt den Cursor ein Zeichen nach links. Tabulatoren und das Zeilenende der vorherigen Zeile werden übersprungen.
CursorCharRight	Bewegt den Cursor ein Zeichen nach rechts. Tabulatoren und das Zeilenende der aktuellen Zeile werden übersprungen.
CursorDown	Cursor eine Zeile nach unten bewegen.
CursorLeft	Cursor eine Spalte nach links bewegen.
CursorRight	Cursor eine Spalte nach rechts bewegen.
CursorSwitchedLeft	Wie *CursorLeft*, berücksichtigt aber Tabulatoreinstellungen (vgl. *SetCursorThroughTabMode*).
CursorSwitchedRight	Wie *CursorRight*, berücksichtigt aber Tabulatoreinstellungen (vgl. *SetCursorThroughTabMode*).
CursorUp	Bewegt den Cursor eine Zeile nach oben
DeleteBlock	Existiert ein gültig markierter Textblock, wird er gelöscht.
DeleteChar	Löscht das Zeichen auf dem der Cursor steht.

Befehlsname	Aktion des Editors
DeleteLine	Löscht die aktuelle Zeile.
DeleteToEOL	Löscht alle Zeichen aus der aktuellen Zeile, es bleibt eine Leerzeile zurück.
DeleteWord	Löscht von der Cursorposition bis zum nächsten Wort.
EndCursor	Bewegt den Cursor ans Ende des Dateipuffers.
ExtendBlockBeg	Steht zu Beginn einer Reihe von Befehlen, die einen Textblock zwischen der Anfangs- und Endeposition des Cursors markieren.
ExtendBlockEnd	Steht am Ende einer Reihe von Befehlen, die mit *ExtendBlockBeg* beginnen.
FixCursorPos	Stellt sicher, daß der Cursorwert eine Zeile zwischen 1 und der Anzahl von Zeilen im Puffer darstellt und daß die Spalte größer als 0 ist. Ist die Option *Cursor through tabs* nicht gesetzt, wird der Cursor nicht in die Mitte eines Tabulators gesetzt.
FixScreenPos	Paßt die Bildschirmanzeige an, um den Cursor sichtbar zu machen.
FullPaintScreen	Zeichnet das gesamte Fenster neu, ohne die alten Inhalte zu übernehmen.
HideBlock	Setzt ein Flag, das anzeigt, daß der ausgewählte Block nicht markiert werden soll.
HighlightBlock	Setzt ein Flag, das bei gültigen Textmarkierungen für die Hervorhebung der markierten Blöcke sorgt.
HomeCursor	Bewegt den Cursor an den Anfang des Dateipuffers.
IndentBlock	Fügt ein Leerzeichen am Anfang jeder Zeile des markierten (ausgewählten) Textes ein.
InsertText	Fügt den literalen String im Puffer an der aktuellen Cursorposition ein. Die Aufrufsyntax lautet *InsertText*(*String*).
LeftOfLine	Bewegt den Cursor an den Anfang der aktuellen Zeile.
LiteralChar	Fügt ein Zeichen an der aktuellen Cursorposition ein, ohne daß Zeichen wie Tabulatoren, Zeichenschaltungen usw. gesondert bearbeitet werden. Die Syntax lautet *LiteralChar*(*c*), wobei *c* ein Zeichen oder ein Integerwert sein kann.
MarkBufModified	Setzt ein Flag, das anzeigt, daß der Inhalt des Puffers nicht mehr mit der entsprechenden Datei auf dem Datenträger übereinstimmt.

Befehlsname	Aktion des Editors
MarkBufUnModified	Löscht ein Flag und zeigt so an, daß von der Identität von Puffer und der Datei auf dem Datenträger ausgegangen werden kann.
MatchPairBackward	Wie *MatchPairForward*. Allerdings wird ausgehend von einem ' oder einem " rückwärts gesucht.
MatchPairForward	Steht der Cursor auf einem der Zeichen (,), {, }, [,] oder auf einem der ersten Zeichen der Zeichenpaare /* oder *,/ wird in der entsprechenden Richtung nach einem passenden Begrenzer gesucht. Steht der Cursor auf einem ' oder ", wird vorwärts gesucht. Der Cursor wird auf die nächste Entsprechung gesetzt, wenn eine gefunden werden kann.
MoveBlock	Wie *CopyBlock*, jedoch wird das Original des ausgewählten Textes gelöscht.
MoveToBlockBeg	Bewegt den Cursor an den Anfang des ausgewählten Textes.
MoveToBlockEnd	Bewegt den Cursor ans Ende des ausgewählten Textes.
MoveToMark	Bewegt den Cursor an die Position, die durch den Befehl *SetMark(n)* gesichert wurde. Verwenden Sie die Syntax *MoveToMark(n)*. *n* ist eine einstellige Zahl zwischen 0 und 9.
MoveToPrevPos	Bewegt den Cursor an die durch »previous position marker« bezeichnete Position.
MoveToTempPos	Bewegt den Cursor zu einer gesicherten temporären Markierung.
Nullcmd	Keine Aktion. Dieser Befehl den Editor auf, führt aber keine Funktion aus. Mit diesem Befehl kann ein Tastendruck ohne Wirkung erzeugt werden.
OutdentBlock	Löscht führende Leerzeichen vom Anfang jeder Zeile des markierten (ausgewählten) Textes.
PageDown	Bewegt den Cursor um die Anzahl der Zeilen im Fenster nach unten.
PageScreenDown	Bildlauf um die Anzahl der Zeilen im Fenster nach unten. Die Cursorposition bleibt unverändert.
PageScreenUp	Bildlauf um die Anzahl der Zeilen im Fenster nach oben. Die Cursorposition bleibt unverändert.
PageUp	Bewegt den Cursor um die Anzahl der Zeilen im Fenster nach oben.

Befehlsname	Aktion des Editors
PaintScreen	Zeichnet das Fenster neu und geht dabei davon aus, daß der Bildschirm die letzten Änderungen durch den Editor richtig dargestellt werden.
ReDo	Führt eine Wiederholung durch. Die genauen Auswirkungen hängen von den Parametereinstellungen ab.
RightOfLine	Bewegt den Cursor an das Ende der Zeile.
RightOfWord	Bewegt den Cursor in die dem Ende eines Wortes folgende Spalte.
ScrollScreenDown	Führt einen Bildlauf um eine Zeile nach unten durch, ohne die Cursorposition zu verändern.
ScrollScreenUp	Führt einen Bildlauf um eine Zeile nach oben durch, ohne die Cursorposition zu verändern.
SetAutoIndent	Schaltet die Option *Auto Indent* ein.
SetAutoOutdent	Schaltet die Option *Backspace Unindents* ein.
SetBlockBeg	Setzt das erste Zeichen des ausgewählten Textes als aktuelle Cursorposition.
SetBlockEnd	Setzt das letzte Zeichen des ausgewählten Textes als aktuelle Cursorposition.
SetCursorThroughTabMode	Schaltet die Option *Cursor Through Tabs* ein.
SetInsertMode	Schaltet den Einfügemodus ein.
SetMark	Setzt eine Markierung auf das Zeichen der aktuellen Cursorposition, so daß ein nachfolgender Befehl *MoveToMark*(*n*) den Cursor wiederherstellen kann. Verwenden Sie die Syntax *SetMark*(*n*). *n* ist eine einstellige Zahl zwischen 0 und 9.
SetOptimalFillMode	Schaltet die Option *Set Optimal Fill* ein.
SetPrevPos	Setzt eine Markierung (»previous position marker«) auf die aktuelle Cursorposition. Diese Markierung wird nur durch einen Aufruf von *SetPrevPos* oder *SwapPrevPos* verändert.
SetTabbingMode	Schaltet die Option *Use Tab Char* ein.
SetTempPos	Sichert die Cursorposition in einer temporären Markierung, die von einigen internen Editorbefehlen verwendet wird. In benutzerdefinierten Makros sollte stattdessen *SetMark* verwendet wirden.
SmartRefreshScreen	Stellt das Fenster neu dar und übergeht dabei alle Teile, die der Editor als unverändert seit der letzten Neudarstellung meldet.

Befehlsname	Aktion des Editors
SmartTab	Fügt Leerzeichen bzw. Tabulatoren entsprechend der Einstellung der Tabulatorbreite in *Use Tab Char* ein.
SwapPrevPos	Tauscht den Wert des Cursors und des »previous position marker« gegeneinander aus.
ToggleAutoIndent	Schaltet den Status der Option *Auto Indent* um.
ToggleAutoOutdent	Schaltet den Status der Option *Backspace Unindents* um.
ToggleCursorThroughTabMode	Schaltet den Status der Option *Cursor Through Tabs* um.
ToggleHideBlock	Schaltet den Status des Flags des markierten (ausgewählten) Textes um.
ToggleInsert	Schaltet den Status der Option *Insert/Overwrite* um.
ToggleOptimalFillMode	Schaltet den Status der Option *Optimal Fill* um.
ToggleTabbingMode	Schaltet den Status der Option *Use Tab Char* um.
TopOfScreen	Bewegt den Cursor in die erste Zeile des aktuell dargestellten Fensters. Die Spalte bleibt unverändert.
UnDo	Führt eine Undo-Operation (Rückgängig machen) durch. Die Auswirkungen hängen von den Parametereinstellungen ab.
WordLeft	Bewegt den Cursor auf den Anfang des vorhergehenden Wortes bzw. das Ende der vorhergehenden Zeile, je nachdem, was zuerst gefunden wird.
WordRight	Bewegt den Cursor auf den Anfang des nächsten Wortes bzw. das Ende der aktuellen Zeile, je nachdem, was zuerst gefunden wird.

Die folgenden Befehle verursachen den Ausstieg aus dem Editor und dürfen deshalb nur als letzter Befehl in einem Makro auftauchen:

Befehlsname	Aktion des Editors
Exit	Verläßt den Editor und aktiviert die Hauptmenüzeile.
GetFindString	Öffnet das Dialogfenster für Such-Operationen.
PrintBlock	Schickt den ausgewählten Text zum Drucker.
ReadBlock	Öffnet ein Dialogfenster, das die Eingabe eines Dateinamens erwartet. Die Datei wird an der aktuellen Cursorposition in den Puffer gelesen und als Textblock markiert.
RepeatSearch	Wiederholt eine Suche mit den vorhergehenden Parametern.

Befehlsname	Aktion des Editors
Replace	Öffnet ein Dialogfenster für die Funktion Replace (Ersetzen).
WordHelp	Öffnet ein Hilfefenster für das Wort unter dem Cursor.
WriteBlock	Öffnet ein Dialogfenster, in dem die Angabe einer Datei erwartet wird, in die dann der ausgewählte Text geschrieben wird.

Jeder der folgenden Befehle entspricht einer Aktion der Tastenkürzel in der IDE. Sie verursachen außerdem den Ausstieg aus dem Editor.

Befehlsname	Aktion des Editors
AddWatch	Fügt einen Watchausdruck hinzu (Strg F7).
ClipClear	Löscht ausgewählten Text (Strg Entf).
ClipCopy	Kopiert ausgewählten Text in die Zwischenablage (Strg Einfg).
ClipCut	Verschiebt ausgewählten Text in die Zwischenablage (Umschalt Entf).
ClipPaste	Schreibt den Inhalt der Zwischenablage an der Cursorposition in den Puffer (Umschalt Einf).
ClipShow	Zeigt die Zwischenablage (kein Tastenkürzel vorgegeben).
CloseWindow	Editorfenster schließen (Alt F3).
CompileFile	Aktuellen Puffer compilieren (Alt F9).
CompileMenu	Öffnet das *Compile*-Menü (Alt C).
DebugMenu	Wählt das *Debug*-Menü (Alt D).
EditMenu	Wählt das *Edit*-Menü (Alt E).
FileMenu	Wählt das *File*-Menü (Alt F).
GotoWindow1	Wählt Fenster #1 (Alt 1).
GotoWindow2	Wählt Fenster #2 (Alt 2).
GotoWindow3	Wählt Fenster #3 (Alt 3).
GotoWindow4	Wählt Fenster #4 (Alt 4).
GotoWindow5	Wählt Fenster #5 (Alt 5).
GotoWindow6	Wählt Fenster #6 (Alt 6).
GotoWindow7	Wählt Fenster #7 (Alt 7).
GotoWindow8	Wählt Fenster #8 (Alt 8).

Befehlsname	Aktion des Editors
GotoWindow9	Wählt Fenster #9 (Alt 9).
Help	Kontextbezogene Hilfe (Strg F1).
HelpMenu	Wählt das *Help*-Menü (Alt H).
Inspect	Prüft Eintrag (Alt F4).
LastHelp	Öffnet vorhergehendes Hilfefenster (Alt F1).
MakeProject	Erstellt Projekt (F9).
Menu	Aktiviert Hauptzeilenmenü (F10).
Modify	Ausdruck auswerten/Variable verändern (Strg F4).
NextError	Geht zum nächsten Eintrag im Meldungsfenster (Alt F8).
NextWindow	Wählt nächstes Fenster in der IDE (F6).
OpenFile	Öffnet Dialogfenster zum Öffnen einer Datei (F3).
OptionsMenu	Wählt das *Options*-Menü (Alt O).
PrevError	Geht zum vorhergehenden Eintrag im Meldungsfenster (Alt F7).
Quit	Verläßt die IDE (Alt X).
ResetProgram	Setzt das getestete Programm zurück (Strg F2).
RunMenu	Wählt das *Run*-Menü (Alt R).
RunProgram	Erstellt und startet das aktuelle Programm (Strg F9).
RunToHere	Führt das Programm bis zur Anweisung unter dem Cursor aus (F4).
SaveFile	Sichert den aktuellen Editorpuffer (F2).
SearchMenu	Wählt das *Search*-Menü (Alt S).
Step	Schrittweise austesten (F8).
SystemMenu	Wählt das *System*-Menü (Alt Spacebar).
Trace	Schrittweise austesten (auch aufgerufene Funktionen) (F7).
ViewCallStack	Zeigt den Aufruf-Stack (Strg F3).
ViewUserScreen	Stellt Anwenderbildschirm dar (Alt F5).
WindowList	Zeigt Fensterliste (Alt 0).
WindowMenu	Wählt das *Window*-Menü (Alt W).
ZoomWindow	Vergrößert/verkleinert das aktuelle Fenster (F5).

Transfer-makros

Die IDE erkennt mehrere Makronamen im Parameter-String des Dialogfensters *Modify/New Transfer Item*. Es gibt drei Arten von Makros: Status, Dateiname und Anweisung. Alle Transfermakros sind weiter unten alphabetisch aufgelistet und werden dort detailliert beschrieben.

Statusmakros

Diese Makros werden entsprechend des aktuellen Status der IDE erweitert. Es gibt folgende Statusmakros:

$COL	$ERRCOL	$INC	$PRJNAME
$CONFIG	$ERRLINE	$LIB	
$DEF	$ERRNAME	$LINE	

Dateinamenmakros

Diese Makros sind Funktionen, die Dateinamen als Argumente haben und verschiedene Teile des Dateinamens zurückliefern. Mit ihnen können Sie neue Dateinamen-Spezifikationen aus bestehenden Dateinamen erstellen. So können Sie z.B. an TDUMP folgendes Makro übergeben:

```
$DIR ($EXENAME) $NAME ($EDNAME).OBJ
```

Dieses Makro erstellt den Verzeichnispfad für die Ausgabe, den Dateinamen (nur für das aktuelle Editorfenster) und eine explizite Namenserweiterung. Wenn Ihr aktuelles Verzeichnis C:\WORK ist, Ihr Ausgabeverzeichnis TEST und im aktiven Editorfenster MYPROG.C steht, erhält TDUMP die Parameter:

```
C:\WORK\TEXT\MYPROG.OBJ
```

Folgende Dateinamenmakros stehen zur Verfügung:

$DIR	$EDNAME	$EXT	$OUTNAME
$DRIVE()	$EXENAME	$NAME()	

Anweisungsmakros

Diese Makros veranlassen die IDE dazu, Aktionen durchzuführen oder Einstellungen zu verändern. Die folgenden Anweisungsmakros stehen zur Verfügung:

$CAP EDIT	$IMPLIB	$PROMPT	$SAVECUR
$CAP MSG(*Filter*)	$MEM(KByte)	$RC	$SAVEPROMPT
$DEP()	$NOSWAP	$SAVEALL	$TASM

$CAP EDIT: Die IDE leitet die Programmausgabe in eine Standarddatei um. Ist das Transferprogramm abgeschlossen, wird ein neues Editorfenster erstellt und die umgeleitete Ausgabe wird angezeigt. Die umgeleitete Ausgabe finden Sie in einem speziellen Editorfenster mit dem Namen *Transfer Output*.

Für $CAP EDIT muß, wenn es korrekt arbeiten soll, das Transferprogramm in die DOS-Standard-Ausgabeeinheit schreiben.

$CAP MSG(Filter): Leitet die Programmausgabe in das Message-Fenster um, verwendet *Filter* wie einen DOS-Filter und konvertiert damit die Programmausgabe in das Message-Fenster-Format. Vier Filter stehen für dieses Makro zur Verfügung:

- GREP2MSG.EXE für GREP,
- IMPL2MSG.EXE für IMPLIB,
- RC2MSG.EXE für den Ressourcencompiler und
- TASM2MSG.EXE für Turbo Assembler (TASM).

Der Quelltext für diese Filter steht Ihnen zu Verfügung, so daß Sie Ihre eigenen Filter für andere Transferprogramme schreiben können. Mit diesem Makro kann jedes Programm mit zeilenorientierten Meldungsausgaben (Datei und Zeilennummer) benutzt werden.

$COL: Spaltenzahl des aktuellen Editors. Ist das aktive Fenster kein Editor, wird der String auf Null gesetzt.

$CONFIG: Kompletter Dateiname der aktuellen Konfigurationsdatei. Ist keine Konfigurationsdatei definiert, steht hier ein Null-String. Dieses Makro ist für Programme gedacht, die die Konfigurationsdatei ansprechen oder verändern. Die Datei wird (nach einer Veränderung) automatisch gespeichert und wieder geladen, wenn die Ablaufsteuerung zur IDE zurückgegeben wird.

Verwenden Sie dieses Makro mit dem Turbo Editor Macro Language (TEML) Compiler. (TEML ist eine Pascal-ähnliche Sprache mit einfachen integrierten Editor-Befehlen; vgl. dazu die Online-Dokumentation.) Damit können Sie die TEML-Makrodatei in einen Editor laden und dann den Turbo Editor Macro Compiler (TEMC) aufrufen, um die Makrodatei zu bearbeiten. Wird die Konfigurationsdatei wieder geladen, werden die neuen oder geänderten Editorbefehle aktiviert. Als Transfer-Wahlpunkt installieren Sie TEMC mit

```
$EDNAME $CONFIG
```

Dies setzt voraus, daß das aktuelle Editorfenster die zu bearbeitende TEML-Makrodatei enthält.

$DEF: Übernimmt den Eintrag aus dem Eingabefeld *Defines* aus dem Dialogfenster *Options/Compiler/Code Generation*. Verwenden Sie dieses Makro, um *define*-Direktiven an einen externen Übersetzer zu spezifizieren.

$DEP(): Mit Hilfe dieses Makros werden Ressourcen als Teil eines Projekt-MAKE neu compiliert, wenn eine der Ressourcen-Komponenten geändert wurde. Dieses Makro wird nur von der Projektverwaltung benutzt.

Nehmen wir beispielsweise an, daß sich die Windows-Ressource MYAPP1.RES

der Anwendung MYAPP1.EXE aus den folgenden Komponenten zusammensetzt:

- MYAPP1.RC (die Ressourcenquelltextdatei)
- MYAPP1.H (die Header-Datei von MYAPP1.EXE)
- MYAPP1.ICO (die Symbol-Ressource für MYAPP1.EXE, eingebunden in MYAPP1.RC)
- MYAPP1.BMP (die Bitmap-Ressource für MYAPP1.EXE, eingebunden in MAYAPP1.RC)

Um sicherzustellen, daß MYAPP1.RES bei Änderungen einer der Komponenten erneut compiliert wird, fügen Sie folgende Abhängigkeiten für MYAPP1.RES in das Feld *Options* im Dialogfenster *Project/Local Options* ein:

```
$DEP (MYAPP1.H MYAPP1.ICO MYAPP1.BMP)
```

Wenn Sie *Compile/Make* aufrufen, liest die Projektverwaltung das Makro $DEP und stellt alle Abhängigkeiten fest, die älter sind als die resultierende MYAPP1.RES-Datei. Sind eine oder mehrere Abhängigkeiten nicht erfüllt, compiliert die Projektverwaltung MYAPP1.RC erneut.

Sie können für jeden Projekteintrag, der von MAKE verwendet wird, explizite Abhängigkeiten vergeben. Dazu müssen Sie nur die Dateien, von denen der Quelltext abhängig sein soll, in runden Klammern, getrennt durch ein Leerzeichen, ein Komma oder einen Strichpunkt angeben. Ist die automatische Abhängigkeitsprüfung eingeschaltet, werden die expliziten Abhängigkeiten nach den automatischen überprüft.

$DIR(): Verzeichnis für das Datei-Argument, kompletter Pfad.

$DRIVE(): Laufwerk des Datei-Arguments in der Form *D:*.

$EDNAME: Kompletter Name der Datei im aktiven Editorfenster. Ist das aktive Fenster kein Editor, ist dies ein Null-String.

$ERRCOL: Spaltennummer des aktuellen Fehlers in der Datei $ERRNAME. Gibt es keine Meldungen, wird der String zum Null-String erweitert.

$ERRLINE: Zeilennummer des aktuellen Fehlers in der Datei $ERRNAME. Gibt es keine Meldungen, wird der String zum Null-String erweitert.

$ERRNAME: Name der Datei, auf die sich die Meldung im Message-Fenster bezieht. Hier steht ein Null-String, wenn es keine Meldungen gibt oder wenn sich die aktuelle Meldung nicht auf die Datei bezieht.

$EXENAME: Name der Programmdatei (einschließlich Ausgabepfad), der sich aus dem Namen des Projekts oder, wenn kein Projekt definiert wurde, aus dem Namen der .EXE-Datei herleitet, die vom aktiven Editorfenster vergeben werden würde. Ist die Windows-Option für DLL-Linker ausgewählt, lautet die Dateinamenserweiterung .DLL.

$EXT(): Namenserweiterung des Datei-Arguments; der Punkt (z.B. .CPP) ist enthalten.

IMPLIB: Dieses Makro führt IMPLIB aus und wird erweitert zu:

```
$NOSWAP $CAP MSG(IMPL2MSG)
$DRIVE ($EXENAME) $DIR ($EXENAME) $NAME ($EXENAME).LIB $EXENA      -
ME|def_name
```

Wenn *Use DLL File exports* im Dialogfenster (*Options/Make*) gewählt ist, wird $EXENAME Bestandteil der Erweiterung. Ist *Use DEF File exports* gewählt, wird der Name der .DEF-Datei im Projekt (dargestellt durch *def_name*) verwendet.

$INC: Dieses Makro übernimmt den Inhalt des Eingabefelds *Include Directories* im Dialogfenster *Options/Directories*.

$LIB: Dieses Makro übernimmt den Inhalt des Eingabefelds *Library Directories* im Dialogfenster *Options/Directories*.

$LINE: Zeilennummer des aktuellen Editorfensters. Ist das aktive Fenster kein Editor, wird der String auf Null gesetzt.

$MEM(*zu reservierende KByte*): Dieses Makro teilt der IDE mit, wieviel Speicher dem Transferprogramm zugewiesen werden soll. Die IDE tritt so viel Speicher ab, wie entweder spezifiziert wurde oder wie maximal verfügbar ist. Wird kein Speicher angefordert, erhalten Sie eine Fehlermeldung.

$NAME(): Dateiname des Arguments; enthält keinen Punkt.

$NOSWAP: Wenn ein Programm läuft, schaltet die IDE nicht in den Benutzerbildschirm um. Es wird ein Fenster geöffnet, in dem der Name des aktuellen Transferprogramms angezeigt ist. Verwenden Sie dieses Makro in Verbindung mit *$CAP*.

$OUTNAME: Das Makro wird zum Pfad und Dateinamem erweitert, der im Eingabefeld *Output Path* (im aktiven Editorfenster) im Dialogfenster *Project/Local Options* steht. Enthält das Projekt z.B. STARS.C, lautet der voreingestellte Pfad im Eingabefeld *Output Path* STARS.OBJ. Befindet sich STARS.C im aktiven Editorfenster, wird $OUTNAME zu STARS.OBJ erweitert. Steht im Eingabefeld `..\MOON.XYZ` , wird $OUTNAME zu ..\MOON.XYZ erweitert. Dieses Makro ist vor allem bei der Spezifizierung von Modulen für benutzerdefinierte Übersetzer nützlich. So könnten Sie beispielsweise einen TLIB-Übersetzer mit folgender Kommandozeile definieren:

```
TLIB MYLIB +$OUTNAME
```

Dadurch wird das Objektmodul der Datei im aktiven Editorfenster der Bibliothek MYLIB hinzugefügt.

$PRJNAME: Die aktuelle Projektdatei. Wurde kein Projekt definiert, wird der String auf Null gesetzt.

$PROMPT: Zeigt den erweiterten Parameter-String in der IDE an, bevor das Transferprogramm aufgerufen wird. Die Kommandozeile, die übergeben werden soll, wird in einem Dialogfenster angezeigt, wo Sie den String erweitern oder ändern können. Die Position des $PROMPT-Makros in der Kommandozeile bestimmt den Inhalt des *Prompt*-Fensters.

Konstante Parameter fügen Sie in die Kommandozeile ein, indem Sie sie vor $PROMPT eingeben, z.B. ist */c* in

```
/c $PROMPT dir
```

konstant und wird im Dialogfenster nicht angezeigt, *dir* dagegen kann editiert werden, bevor das Kommando ausgeführt wird.

$RC: Dieses Makro ist bereits definiert und wird zusammen mit dem Ressourcencompiler verwendet. Da dieser Compiler für zwei ganz unterschiedliche Aufgaben aufgerufen werden kann, wird $RC unterschiedlich erweitert. Die Erweiterung hängt davon ab, ob eine .RC-Datei zu einer .RES-Datei compiliert oder ob die .RES-Datei in eine ausführbare Datei gebunden werden soll. Führen Sie dementsprechend folgende Änderungen durch:

- Soll eine .RC-Datei compiliert werden, müssen Sie die Kommandozeile in *Project/Local Options* für die .RC-Datei in der Projektverwaltung ändern.
- Soll eine .RES-Datei zu einer .EXE- oder .DLL-Datei gebunden werden, müssen Sie die Optionen in *Options/Transfer/Resource Compiler* ändern.

Für das Compilieren einer .RC- zu einer .RES-Datei wird $RC folgendermaßen erweitert:

```
$SAVE CUR $NOSWAP $CAP MSG(RC2MSG) -R -I$INC -FO $OUTNAME $EDNAME
```

Für das Einbinden einer .RES- in eine .EXE-Datei wird $RC folgendermaßen erweitert:

```
$NOSWAP $CAP MSG(RC2MSG) res_name$EXENAME
```

Die Variable *res_name* wird in folgender Reihenfolge definiert:

1. Ist in dem Projekt eine Datei mit der Namenserweiterung .RES vorhanden, wird diese Datei verwendet.

2. Gibt es keine Datei mit der Namenserweiterung .RES *und* ist eine Datei mit der Namenserweiterung .RC vorhanden, dann ist *res_name* der in $OUTNAME vergebene Name für die .RC-Datei.

3. Trifft keine der beiden obigen Bedingungen zu (d.h. daß keine Ressourcen vorhanden sind), bleibt *res_name* leer.

$SAVE ALL: Sichert - ohne Abfrage - alle Dateien in allen Editorfenstern, die verändert wurden.

$SAVE CUR: Sichert die Datei im aktuellen Fenster, wenn sie verändert wurde. Dadurch wird gewährleistet, daß immer die aktuelle Version der Quelltextdatei verwendet wird.

$SAVE PROMPT: Fragt nach, ob eine Sicherung erfolgen soll, wenn Dateien aus den Editorfenstern nicht gesichert wurden.

$TASM ist für Turbo Assembler vordefiniert. Es verwendet den Filter TASM2MSG, um TASM-Meldungen zu bearbeiten. $TASM ist im Grunde genommen eine Abkürzung für `$NOSWAP $SAVE CUR $CAP MSG(TASM2MSG) $EDNAME,$OUTNAME` .

WRITEMSG(Dateiname) kopiert den Inhalt des Message-Fensters in die angegebene ASCII-Datei. Der Übersetzer kann diese Datei parsen und entsprechend der Meldungen handeln. So schreibt `WRITEMSG(C:\MESSAGES.TXT)` die Datei MESSAGES.TXT in das Stammverzeichnis.

Aufrufen von DOS-Kommandos

Wenn Sie in der IDE DOS-Kommandos aufrufen wollen, können Sie dafür ein einfaches Transfermakro schreiben:

```
command /c $MEM(128) $PROMPT
```

Wenn Sie diesen Transfereintrag aufrufen, wird ein Dialogfenster mit der Aufforderung zur Eingabe eines DOS-Kommandos angezeigt. Da der Befehl $PROMPT weiter hinten in dem String steht, erscheint der Text *command /c* nicht im Eingabefeld. Sie können sofort z.B. `dir`, `chdsk`, `del *.*` oder ein anderes DOS-Kommando eingeben.

Einstellung des Transferspeichers

Unterschiedliche Programme haben verschiedenen Speicherbedarf. GREP braucht beispielsweise im Vergleich zu vielen bekannten Editoren, die oft mit 200-300 KByte arbeiten, sehr wenig Hauptspeicher.

Mit dem $MEM()-Makro können Sie angeben, wieviel Speicher die IDE dem Transferprogramm zur Verfügung stellen soll. Je weniger Speicher zugewiesen wird, desto schneller erfolgt der Transfer.

Kann die IDE nicht den ganzen angeforderten Speicher zur Verfügung stellen, vergibt sie soviel wie möglich. Beim Testen mit dem Debugger z.B. wird sehr viel Speicher benötigt. Mit *Program Reset* (Strg F2) geben Sie den vom Debugger belegten Speicher frei.

Soll der ganze Speicher freigegeben werden, tragen Sie einfach eine große Zahl wie z.B. 640 KByte ein. Wieviel Speicher aktuell von Turbo C++ abgetreten werden kann, hängt in erster Linie davon ab, wieviel Speicher beim Start von Turbo C++ zur Verfügung stand.

Index

A
Abgeleitete Klassen (C++) 111
Abgeleiteter Typ 40
Ablaufkontrolle 97
Addition (Zeigerarithmetik) 58
Additionsoperator + 89
Additive Operatoren
  + (Addition) 89
  - (Subtraktion) 89
adjustfield, ios-Datenelement 187
Adreßoperator & 86
allocate
  streambuf-Elementfunktion 195
ANSI-C-Implementation 235
Anweisungen 97
  Ablaufkontrolle 97
  Ausdruck 99
  Auswahl 99
  Block 98
  Label 99
  Sprünge 103
  Sprungziele 99
  Wiederholung 101
Anweisungsmakros 391
Argumente
  in Strings konvertieren 160
  variable Anzahl 22
Argumentkonvertierung
  (Funktion) 65
Arithmetik 201
Arrays 59
  deklarieren 59
  Indizierung 83
  initialisieren 45
  Literale in 45
  mehrdimensionale 60
AT&T C++ Version 2.1 3
attach
  filebuf-Elementfunktion 184
  fstreambase-Elementfunktion 186
Attribute
  von Objekten 27
Aufzählungstypen 73
  Konstanten 17
Ausdrücke
  Anweisungen 99
  Auswertungsreihenfolge 79
  Berechnungen 75
  Fehler 80
  in C++ 79
  Turbo C++ 76
Ausgabe
  Streams 175, 177
Ausrichtung
  Strukturen und 69
Auswahlanweisungen 99
  if 100
  switch 100
Auswertungsreihenfolge
  von Ausdrücken 79
auto (Speicherklasse) 30
auto (Speicherklassen-
  Spezifizierer) 38, 46
automatic (Speicherklasse) 211

B
Backslash 14
bad
  ios-Elementfunktion 188
base
  streambuf-Elementfunktion 195
basefield, ios-Datenelement 187
Basisklassen (C++) 111
  virtuelle 122
  Zugriffskontrolle 119
BASM (integrierter Assembler) 207
BCD-Arithmetik 205
  Konvertierung 206

Rundung von Zahlen 206
bcd.h (Deklarationsdatei) 205
Bedingungsoperator ? 94
Berechnung
von Ausdrücken 75
Bezeichner 9
Dauer 29
Deklaration 26
Groß-/Kleinschreibung 10
Gültigkeitsbereich 10, 27
Längenbeschränkung 10
pascal 10
pascal (Modifizierer) 51
Sichtbarkeit 28
signifikante Zeichenzahl 10
verborgene 28
Bibliotheksverwaltung
Fehlermeldungen 251
Binäre Operatoren
überladen 138
Bindung 31
externe 47
interne 47
bitalloc
ios-Elementfunktion 188
Bitfelder 70
Bitweiser exklusiver ODER-Operator ^ 93
Bitweiser inklusiver ODER-Operator | 93
Bit-Komplement-Operator ~ 87
Bitweiser UND-Operator & 92
blen
streambuf-Elementfunktion 195
Block
einschließender 30
Gültigkeitsbereich 27
Blockanweisung 98
Liste 99
bp, ios-Datenelement 187
break-Anweisung 103

C

C++
Operatoren 96
Programme 105
Streams 173
Version 2.1 von AT&T 3
C++ Gültigkeitsbereich-Zugriffsoperator :: 107
cdecl
Modifizierer 51
$CAP EDIT-Transfermakro 392
$CAP MSG-Transfermakro 392
CDECL (manifeste Konstante) 169
char (Typ) 41
clear
ios-Elementfunktion 188
close
filebuf-Elementfunktion 184
fstreambase-Elementfunktion 186
$COL-Transfermakro 392
Compilierung
bedingte 157, 163
$CONFIG-Transfermakro 392
const (Modifizierer) 48, 57
constrea.h 183
continue-Anweisung 103
cplusplus (manifeste Konstante) 169

D

DATE (manifeste Konstante) 169
Dateien
einbinden 161
Gültigkeitsbereich 28
Öffnen-Modus 181
Streams 181
Dateinamenmakros 391
Dauer 29
Bezeichner 29
dynamische 30
Gültigkeitsbereich und 30
lokale 30
statisch 29
von Objekten 29
dec (Manipulator) 178
DEFAULTS.TEM 381f
#define-Makro 156
defined 164
Definitionen
von Funktionen 63
Definitionsdeklarationen 26
$DEF-Transfermakro 392
Deklarationen 25
Array 59
Bezeichner 26
einfache 46
externe 31, 38
formale Parameter 64
komplexe 53
Objekt 25
Prototyp 61
Referenz (C++) 59
Speicherklassen 38

Strukturelemente 67
Strukturen 66
tentative Definition 33
unvollständige 70
Varianten 73
von Funktionen 60
Zeiger 56
Deklarationsdateien 153
Deklarationsliste 64
Deklarationssyntax 33f
Deklaratoren 53
-- Dekrement-Operator 85
delete (C++ Speicherfreigabe-operator) 108
$DEP-Transfermakro 392
Destruktoren (C++) 124, 131
abort und 132
aufrufen 132
exit und 132
virtuelle 133
Dezimalkonstanten 12
Dezimalzahlen
Umwandlung in Binärzahlen 205
Direktiven
Präprozessor 153
Präprozessor (formale Definition) 154
$DIR-Transfermakro 393
Divisionsoperator / 88
DLL (manifeste Konstante) 170
do while-Anweisung 102
doallocate
strstreambuf-Elementfunktion 198
DOS-Kommandos aufrufen 396
double (Gleitkommatyp) 43
double (Typ) 41
$DRIVE-Transfermakro 393
Dynamische Dauer 30
Dynamische Linkbibliothek (.DLL) 228
anlegen 230
Speicher 231
statische Daten 232
Zeiger 231
Dynamischer Speicher 30

E

eback
streambuf-Elementfunktion 195
ebuf
streambuf-Elementfunktion 195
Eckige Klammern 20
Editormakros 381
$EDNAME-Transfermakro 393
egptr
streambuf-Elementfunktion 195
Einfacher Zuweisungs-operator = 95
Eingabe
benutzerdefinierte Typen 180
Streams 179
Einschließender Block 30
Elementdeklarationen 67
Elemente (C++)
Gültigkeitsbereich 116
qualifizierte Namen 117
statische 115
Zugriffskontrolle 118
Elementfunktionen (C++) 113
Destruktoren 131
Konstruktoren 125
Elementliste (C++) 113
#elif 163
Ellipsen 22, 64
#else 163
EMU.LIB 201
#endif 163
endl (Manipulator) 178
ends (Manipulator) 178
enum (Aufzählungstyp) 73
eof
ios-Elementfunktion 188
Epilog 221
epptr
streambuf-Elementfunktion 195
$ERRCOL-Transfermakro 393
$ERRLINE-Transfermakro 393
$ERRNAME-Transfermakro 393
#error 166
Escape-Sequenzen 14f
$EXENAME-Transfermakro 393
_export (Schlüsselwort) 222, 224
Export-Funktionen 224
extern (Speicherklasse) 30
extern (Speicherklassen-Spezifizierer) 47, 63
Externe Bindung 31
Externe Deklarationen 31
$EXT-Transfermakro 393

F

fail
ios-Elementfunktion 188
far (Modifizierer) 52
fd
filebuf-Elementfunktion 184

Fehler
  Ausdrücke und 80
  C++ und 109
Fehlermeldungen 247
  Bibliotheksverwaltung 251
  Laufzeit 250
  Linker 251
FILE (manifeste Konstante) 170
filebuf (Klasse) 183
fill
  ios-Elementfunktion 188
flags
  ios-Elementfunktion 188
float (Gleitkommatyp) 42
float (Typ) 41
floatfield, ios-Datenelement 187
flush
  ostream-Elementfunktion 193
flush (Manipulator) 178
for-Anweisung 102
Formatierung
  Streams 176
Formatstatus-Flags 176
Fortsetzungszeile 6
forward-Referenz 26
FP87.LIB 202
freeze
  strstreambuf-Elementfunktion 198
Friend-Funktionen (C++) 122
friend-Spezifizierer (C++) 113, 122
fstream (Klasse) 185
fstreambase (Klasse) 185
Füllzeichen 178
Funktiondefintionen 63
Funktionen 60
  aufrufen 65
  deklarieren 60
  Gültigkeitsbereich 27
  Interrupt- 213
  main 60
  Name 64
  Strukturen und 67
  virtuelle (C++) 140
  Zeiger auf 55
Funktions-Template 145
Funktionsaufruf-Operator ( ) 84
Funktionsprototyp
  Gültigkeitsbereich 27

G

gbump
  streambuf-Elementfunktion 195
gcount
  istream-Elementfunktion 190
Genauigkeit
  von Dezimalzahlen 205
Geschweifte Klammern 21
get
  istream-Elementfunktion 191
getline
  istream-Elementfunktion 191
Gleichheitsoperatoren 91
  != (Ungleichheit) 92
  == (Gleichheit) 92
Gleitkomma
  Konstanten 16
  Optionen 201
  Schnellversion 202
Gleitkomma-Code
  Register und 203
Gleitkomma-Routinen
  Emulation 201
  Fehlerbedingungen 203
Gleitkommatypen 42
  double 43
  float 42
Globale Bezeichner
  Präprozessor und 158
good
  ios-Elementfunktion 188
goto-Anweisung 104
gptr
  streambuf-Elementfunktion 195
Grammatik 2
  lexikalische 5
  semantische 25
Groß-/Kleinschreibung
  Bezeichner und 10
Gültigkeitsbereich 27, 112, 116
  Bezeichner und 10, 27
  Block 27
  Datei 28
  Dauer und 30
  Funktion 27
  Funktionsprototyp 27
  Klasse 28
  und #include 161
  und Strukturen 69
Gültigkeitsbereich-
  Zugriffsoperator :: (C++) 107

H

Header-Dateien 153
Heap 30

hex (Manipulator) 178
Hexadezimale Zahl 14
hexadizimale Konstanten 13
Hilfe
  Anwenderklassifizierung 336
  beenden 377
  Dateistruktur 339ff
  Fehlermeldungen 249
  Grafikbilder 344
  kontextabhängige 339, 372ff
  Planung 336
  Schlüsselwörter 349
  Schlüsselworttabelle 376
  Steuercodes 346
Hilfe-Compiler 369f
Hilfe-Projektdatei 360f, 366f, 369
  alternative Kontextstrings 366
  Bitmaps, referenzierte 369
  Build-Option 362
  compilieren 369
  Compress-Option 366
  erstellen 359
  Forcefont-Option 364
  Index-Option 363
  kontextabhängige Hilfe 372ff
  kontextabhängige Themen 367
  Mapfontsize-Option 365
  Multikey-Option 365
  Root-Option 363
  Schlüsselworttabelle, Zugriff auf 376
  Title-Option 364
  Warning-Option 361
  Zugriff von Anwendung aus 371
Hilfegrafiken (Bitmaps) 354ff
Hilfeindex 337
Hilfeprotokoll 357
Hilfesystem
  Anwender und 334
  Aufruf von WinHelp 371ff
  Entwicklungszyklus 333ff
  Grafikbilder 344
  Programmierer und 335
Hilfetext
  Layout 342f
  Schriften 343
huge (Modifizierer) 52

I

I/O-Stream-Bibliothek 173
#if 163
if-Anweisung 100
#ifdef 157, 164
#ifndef 157, 164
ifstream (Klasse) 186
  Konstruktor 181
ignore
  istream-Elementfunktion 191
Implementation (ANSI C) 235
$IMPLIB-Transfermakro 394
IMPLIBW 229
Importbilbiotheken
  anlegen 229
  IMPLIB-Programm ausführen 394
in_avail
  streambuf-Elementfunktion 195
#include 161
Include-Dateien 153
  suchen 162
$INC-Transfermakro 394
Indirektion 23
Indirektionsoperator * 86
init
  ios-Elementfunktion 188
Initialisieren 44
  Anfangswert 44
  Array 45
  Strukturen 45
  Varianten 45
Inkrement-Operator ++ 84
Inline-Ausdehnung (C++) 114
Inline-Funktionen (C++) 114
int (Typ) 41
Integerkonstanten 12
Integertypen 42
Interne Bindung 31
Interpunktionszeichen 20
interrupt (Modifizierer) 50
Interrupt-Funktionen 213
Interrupt-Vektoren 50, 213
ios (Klasse) 174, 187
ios-Datenelement 187
iostream (Klasse) 190
iostream_withassign (Klasse) 190
is_open
  filebuf-Elementfunktion 184
istream (Klasse) 190
  abgeleitete Klassen von 181
istream_withassign (Klasse) 192
istrstream (Klasse) 192

K

Kapitelübersicht 1
Klammern
  eckige 20

geschweifte 21
runde 21
Klassen
Streams (C++) 183
Klassen (C++) 111
abgeleitete 111
Basisklasse 111
Elementfunktionen 113
Elementliste 113
Gültigkeitsbereich 28, 112
initialisieren 129
Namen 111
Objekte von 113
Typen 111
Klassendeklarationen 38
Klassenelemente (C++) 113
Klassenobjekte (C++) 113
Komma-Operator , 96
Kleiner-als-Operator < 90
Kleiner-gleich-Operator <= 91
Kommentar 6
für asm-Anweisungen 208
in C 6
in C++ 7
verschachtelter 7
Konstanten 10
Aufzählungstyp 17
dezimal 12
Escape-Sequenz 14
Gleitkomma- 16
hexadezimale 13
Integer- 12
interne Darstellung 20
mit zwei Zeichen 15
oktale 13
String 17
und Vorzeichen 15
Wertebereiche 20
Wide character 16
Zeichen 14
Zeiger und 57
Konstantenausdruck 20
Konstruktoren
filebuf 183
fstream 185
fstreambase 185
ifstream 181, 186
ios 187
iostream 190
iostream_withassign 190
istream 190
istream_withassign 192
istrstream 192
ofstream 181, 192
ostream 193
ostream_withassign 194
ostrstream 194
streambuf 195
strstream 199
strstreambase 197
strstreambuf 198
Konstruktoren (C++) 124f
Aufrufreihenfolge 127
für globale Variable 125
Kopierkonstruktor 126
lokale Objekte 125
Standard 126
überladen 127
und Initialisierung 129
Konsument 173
Konvertierungen
besondere 44
Referenztypen (C++) 107
Standard 43
von BCD-Zahlen 206
von Zeigern 59
Kopierkonstruktor (C++) 126
Koprozessor (Emulation) 201

L

l-Suffix 14
L-Wert 26
Labels und Maschinensprache 213
Länge
Bezeichner 10
Laufzeit-Fehlermeldungen 250
Lebensdauer
Objekt 27
Lexikalische Grammatik 5
LibMain (MS-Windows-Funktion) 230
$LIB-Transfermakro 394
LINE (manifeste Konstante) 170
#line 165
$LINE-Transfermakro 394
Linker
Fehlermeldungen 251
Linksschiebeoperator << 90
_loadds (Modifizierer) 53
Logische Negation ! 87
Logischer ODER-Operator || 94
Logischer UND-Operator && 93
Lokale Dauer 30
long (Typ) 41
long-Suffixe 13

M
MACROS.TEM 381
main (Funktion) 60, 167
Makroerweiterung 156
Makros
  Anweisungsmakros 391
  Dateinamenmakros 391
  Editor- 381
  Parameter 158
  Statusmakros 391
  stdarg.h 62
  Transfermakros 391
  vordefinierte 169, 171
Manifeste Konstanten 169
Manipulatoren 177
Maschinenbefehle
  in C-Programmen 208
Maschinenprogramme
  Zugriff auf Strukturelemente 212
Maschinensprache
  indirekte Sprünge 213
  Opcodes 209
  Sprungbefehle und Labels 213
  Zugriffe auf Turbo C++ 211
Mathematischer Koprozessor 201
Mehrdimensionale Arrays 60
$MEM-Transfermakro 394, 396
Microsoft Windows
  Compilieren in der IDE 217
  Dynamische Linkbibliothek 228
  Epilog 221
  Importbibliothek 229
  LibMain (Funktion) 230
  Linken in der IDE 217
  Moduldefinitionsdatei 219, 225
  Prolog 221
  Ressourcencopmiler 219
  Ressourcendateien 219
  WEP (Funktion) 230
  WinMain (Funktion) 221
Microsoft Windows DLL: alle
  Funktionen exportierbar 223
Microsoft Windows DLL: explizit
  exportierte Funktionen 223
Microsoft Windows: alle
  Funktionen exportierbar 222
Microsoft Windows: explizite
  Funktionen exportiert 222
Microsoft Windows: intelligente Call-
  backs 223
Microsoft-Windows-Anwendung
  s. Microsoft Windows
Minus-Operator - 87
Modifizierer 48
  _loadds 53
  _saveregs 53
  cdecl 51
  const 48, 57
  far 52
  Funktionsdefinition und 63
  für Funktionen 52
  huge 52
  interrupt 50
  near 52
  pascal 51
  volatile 50
Moduldefinitionsdatei 219, 225
  .DEF-Datei 219
  IMPORTS-Bereich 229
Modulo-Operator % 88
MSDOS (manifeste Konstante) 170
Multiplikationsoperator * 88
Multiplikative Operatoren 88
  % (Modulo) 88
  * (Multiplikation) 88
  / (Division) 88

N
Namen (s. Bezeichner)
  ergänzen 32
$NAME-Transfermakro 394
Namensbereich
  Strukturen und 69
Namensergänzung 32
near (Modifizierer) 52
new (C++ Speicherbelegungs-
  operator) 108
new-Handler 109
$NOSWAP-Transfermakro 394
Null-Anweisung 99
Null-Direktive 155
Null-Zeichen
  String 17

O
Objekte
  Adressen 55
  automatisch 30
  Deklaration 25
  Lebensdauer 27
  mit lokaler Dauer 30
  mit statischer Dauer 29
  Position im Speicher 27
  Zeiger auf 55

oct (Manipulator) 178
ofstream (Klasse) 192
  Konstruktor 181
Oktale Konstanten 13
Oktale Zahl 14
Opcodes 209
open
  filebuf-Elementfunktion 184
  fstream-Elementfunktion 185
  fstreambase-Elementfunktion 186
  ifstream-Elementfunktion 187
  ofstream-Elementfunktion 193
operator (C++ Schlüsselwort) 134
Operator-Funktionsname (C++) 134
Operatoren
  additive 89
  Bedingungs- 94
  C++ 96
  delete 108
  exklusiver ODER-Operator ^ 93
  Gleichheit 91
  inklusiver ODER-Operator | 93
  Komma-Operator , 96
  logischer ODER-Operator || 94
  logischer UND-Operator && 93
  multiplikative 88
  new 108
  Präfix 83, 85
  relationale 90
  Schiebeoperatoren 89
  sizeof 87
  Suffix 83
  überladen 79, 134
  unäre 85
  UND-Operator & 92
  Zuweisung 95
Operatorfunktionen (C++) 135
ostream (Klasse) 193
  abgeleitete Klassen von 181
ostream_withassign (Klasse) 194
ostrstream (Klasse) 194
$OUTNAME-Transfermakro 394
out_waiting
  streambuf-Elementfunktion 196
overflow
  filebuf-Elementfunktion 184
  strstreambuf-Elementfunktion 198

P

Parameter
  Deklarationsliste 64
  deklarieren 64
  formale 64
  Makros und 158
  Standardwert 23
  variable Anzahl 22
PASCAL (manifeste Konstante) 170
pascal (Modifizierer) 51
pbase
  streambuf-Elementfunktion 196
pbump
  streambuf-Elementfunktion 196
pcount
  ostrstream-Elementfunktion 194
peek
  istream-Elementfunktion 191
Plus-Operator + 87
pptr
  streambuf-Elementfunktion 196
Präfix-Operatoren 83
  ++ (Inkrement) 85
  -- (Dekrement) 85
#pragma 166f
#pragma argsused 167
#pragma exit 167
#pragma hdrfile 168
#pragma hdrstop 168
#pragma saveregs 168
#pragma startup 167
Präprozessor 153
  Direktiven 5, 153
Präprozessor-Direktiven 23, 154
  Syntax 153
precision
  ios-Elementfunktion 188
$PRJNAME-Transfermakro 394
Produzent 173
Programme
  in C++ 105
Programmierhandbuch
  Kapitelübersicht 1
Prolog 221
$PROMPT-Transfermakro 395
Prototyp (Funktion) 61
Pseudovariable 9
public (Zugriffs-Spezifizierer) 73
put
  ostream-Elementfunktion 193
putback
  istream-Elementfunktion 191

R

R-Wert 26
$RC-Transfermakro 395

rdbuf
fstream-Elementfunktion 185
fstreambase-Elementfunktion 186
ifstream-Elementfunktion 187
ios-Elementfunktion 189
ofstream-Elementfunktion 193
strstreambase-Elementfunktion 197
rdstate
ios-Elementfunktion 189
read
istream-Elementfunktion 191
Referenzdeklarationen 26
Referenzdeklarationen (C++) 59
Referenzdeklarator (C++) 105
Referenzen (C++) 105
als Argumente 106
einfache 105
Referenztypen (C++) 59, 105
Konvertierung 107
register
des Koprozessors 203
Speicherklasse 30
Zuordnung 211
register (Speicherklassen-Spezifizierer) 38, 47
Registervariable 9
Registerwerte 53
Relationale Operatoren 90f
< (Kleiner) 90
<= (Kleiner gleich) 91
Reservierte Wörter
auto 211
resetiosflags (Manipulator) 178
Ressourcencompiler 219
Ressourcendateien 219
Ressourcen 219
return-Anweisung 104
Rückgabetyp
Funktionsdefinition 63
Runde Klammern 21
Rundung
von BCD-Zahlen 206

S

$SAVE ALL-Transfermakro 395
$SAVE CUR-Transfermakro 395
$SAVE PROMPT-Transfermakro 395
_saverege (Modifizierer) 53
sbumpc
streambuf-Elementfunktion 196
schachteln
von Makros 156
Schiebeoperatoren 89
< (Links) 90
> (Rechts) 90
Schlüsselwörter 8
seekg
istream-Elementfunktion 191
seekoff
filebuf-Elementfunktion 184
streambuf-Elementfunktion 196
strstreambuf-Elementfunktion 198
seekp
ostream-Elementfunktion 193
seekpos
streambuf-Elementfunktion 196
Semantische Grammatik 25
setb
streambuf-Elementfunktion 196
setbase (Manipulator) 178
setbuf
filebuf-Elementfunktion 184
fstreambase-Elementfunktion 186
streambuf-Elementfunktion 196
strstreambuf-Elementfunktion 198
setf
ios-Elementfunktion 189
setfil (Manipulator) 178
setg
streambuf-Elementfunktion 196
setiosflags (Manipulator) 178
setp
streambuf-Elementfunktion 196
setprecision (Manipulator) 178
setstate
ios-Elementfunktion 189
setw (Manipulator) 178
sgetc
streambuf-Elementfunktion 196
sgetn
streambuf-Elementfunktion 196
short (Typ) 41
Sichtbarkeit 28
signed (Typ) 41
sizeof-Operator 87
snextc
streambuf-Elementfunktion 197
Speicherklassen 30
automatic 211
register 211
Typ 27
Speicherklassen-Spezifizierer 46
auto 38, 46
extern 47

Funktionsdefinition und 63
register 38, 47
static 47
Speichermodelle 169
Sprunganweisungen 103
break 103
continue 103
goto 104
return 104
Sprungziele
Label 99
sputback
streambuf-Elementfunktion 197
sputc
streambuf-Elementfunktion 197
sputn
streambuf-Elementfunktion 197
Stack
Variable 30
Standardbibliothek 26
Standardkonstruktor (C++) 126
state, ios-Datenelement 187
static
Speicherklasse 30
Speicherklassen-Spezifiziere) 47
static (Speicherklassen-Spezifizierer) 115
Statische Dauer 29
Statusmakros 391
stdarg.h (Makros) 62
STDC (manifeste Konstante) 170
Steuercode 14
stossc
streambuf-Elementfunktion 197
str
ostrstream-Elementfunktion 194
strstream-Elementfunktion 199
strstreambuf-Elementfunktion 198
streambuf (Klasse) 174, 195
abgeleitete Klassen 174
Streams 173
Ausgabe 175, 177
Datei 181
Eingabe 179
Formatierung 176
Manipulatoren 177
String-Streams (C++) 182
Stringkonstanten 17
Strings
Null-Zeichen 17
Zeilentrennung und 18
strstream (Klasse) 199
strstreambase (Klasse) 197
strstreambuf (Klasse) 198
struct (Schlüsselwort) 66
Strukturen 65
Ausrichtung 69
deklarieren 66
Elemente 67
Elementzugriff 68
Funktionen und 67
initialisieren 45
Namensbereich 69
struct-Schlüsselwort 66
unbenannte 66
Zugriffe 212
Struktur-/Varianten-Element-operator . 84
Struktur-/Varianten-Zeiger-operator -> 84
Subtraktion
Zeigerarithmetik 58
Subtraktionsoperator - 89
Suffix
long 13
unsigned 13
Suffix-Operatoren 83
( ) (Funktionsaufruf) 84
++ (Inkrement) 84
-- (Dekrement) 85
switch-Anweisung 100
Symbole 2, 5, 8
verschmelzen 160
sync
filebuf-Elementfunktion 184
sync_with_stdio
ios-Elementfunktion 189
Syntax 3
Präprozessor-Direktiven 153

T

Tag-Name 66
$TASM-Transfermakro 395
TCPLUSPLUS (manifeste Konstante) 171
tellg
istream-Elementfunktion 192
tellp
ostream-Elementfunktion 194
TEMC 381f
vordefinierte Editorbefehle 384
TEML (Turbo Editor Macro Language) 381
Templates 144

Argumente 149
explizite Funktionen 147
Funktions-Template 145
implizite Funktionen 147
Klassen 148
Objekte 150
spitze Klammern 149
typensichere Listen 149
Zeiger 150
TEMPLATES (manifeste Konstante) 171
Tentative Definition 33
Terminologie 3
Themen
Dateistruktur 339ff
Definitionen 353f
Inhalte 337
kontextabhängige 339
Kreuzreferenzen 353
Sprünge 353
Struktur 337
Themendateien
Beispiele 379ff
Definitionen 353f
Kontextstrings 348
Kreuzreferenzen 353
Schlüsselwörter 349
Sprünge 353
Steuercodes 346
Such-Sequenzen 351ff
symbolische Namen 347ff
Textprogramm 345
Titel-Fußnoten 348
überwachen 357
verwalten 356
this (C++ Schlüsselwort) 113
tie
ios-Elementfunktion 189
TIME (manifeste Konstante) 171
Token 5, 8
Transfermakros 391
Transferprogramme
Speicherzuweisung 394
Transferspeicher 396
Trennzeichen 20
Turbo C++ (Ausdrücke) 76
TURBOC (manifeste Konstante) 171
Typ-Spezifizierer 40
Typ-Taxonomie 40
typedef (Schlüsselwort) 47
Typen
char 41
double 41
float 41
Gleitkomma 42
grundlegende 41
int 41
Integer 42
long 41
Objekt 25
short 41
signed 41
Speicherklasse 27
unsigned 41
void 41
Typumwandlung
von Zeigern 59

U

u-Suffix 14
Übergabe-Operator 175
Überladen (C++)
-> 139
binäre Operatoren 138
delete 136
Funktionsaufruf-Operator () 139
Index-Operator 139
Konstruktoren 127
new 136
unäre Operatoren 137
Zuweisungsoperator = 138
Überladene Operatoren (C++) 134
vererben 136
Überlauf
Ausdrücke und 80
Übernahme-Operator 179
Übersetzungseinheit 30
Unäre Operatoren
! (Logische Negation) 87
überladen 137
& (Adreßoperator) 86
* (Indirektion) 86
+ (Plus-Operator) 87
- (Minus) 87
~ (Bit-Komplement) 87
unbuffered
streambuf-Elementfunktion 197
#undef (Direktive) 156
underflow
filebuf-Elementfunktion 184
strstreambuf-Elementfunktion 199
Ungleich-Operator != 92
union
siehe Varianten

unsetf
  ios-Elementfunktion 189
unsigned (Typ) 41
unsigned-Suffixe 13
Unvollständige Deklarationen 70

V
Varainten
  initialisieren 45
Variable
  Register- 9
Variablendeklarationen 46
Varianten 71
  Deklarationssyntax 73
Vektoren für Interrupts 213
Verborgen
  doppelte Bezeichner 28
Vererbung (C++)
  überladene Operatoren 136
Vergleich
  Zeigerarithmetik 58
virtual (C++ Spezifizierer) 141
Virtuelle Basisklassen (C++) 122
Virtuelle Destruktoren (C++) 133
Virtuelle Funktionen (C++) 140
  abgeleitete Klassen 140
  Funktionstyp ändern 140
  virtual 141
void (Typ) 41
volatile (Modifizierer) 50

W
WEP (MS-Windows-Funktion) 230
whilw (Anweisung) 101
Whitespace 5
  und Strings 18
Wide character-Konstanten 16
width
  ios-Elementfunktion 189
Wiederholungsanweisungen 101
  do while 102
  for 102
  while 101
Windows DLL: alle Funktionen
  exportierbar 223
Windows DLL: explizit
  exportierte Funktionen 223
Windows: alle Funktionen
  exportierbar 222
Windows: explizite Funktionen
  exportiert 222
Windows: intelligente Callbacks 223
_Windows (manifeste Konstante) 171
WinMain (MS-Windows-Funktion) 221
write
  ostream-Elementfunktion 194
$WRITEMSG-Transfermakro 396
ws (Manipulator) 178

X
x_fill, ios-Datenelement 187
x_flags, ios-Datenelement 187
x_precision, ios-Datenelement 187
x_tie, ios-Datenelement 187
x_width, ios-Datenelement 187
xalloc
  ios-Elementfunktion 190

Z
Zeichen
  mit zwei Bytes 15
Zeichenkonstante 14
Zeiger 54
  auf Funktionen 55
  auf Objekte 55
  berechnen 58
  const-Modifizierer 57
  Deklarationen 56
  Größe 55
  Konstanten und 57
  Konvertierungen 59
  Zuordnungsrestriktionen 56
Zeigerarithmetik 58
Zeilenfortsetzung 160
Zeilennummer 165
Zeilentrennung 6
  Strings und 18
Zugriffskontrolle (C++)
  Basisklassen 119
  Elemente (C++) 118
  private 118
  protected 118
  public 118
Zuweisungsoperatoren 95
  = (einfacher) 95
  kombinierte 96

PROGRAMMIER-
HANDBUCH

BORLAND® C++

BORLAND

BORLAND GMBH, GAUTINGER STR. 10,POSTFACH 1209, D-8130 STARNBERG, TELEFON 0 81 51 / 263-0, TELEFAX 0 81 51 / 142 09
CORPORATE HEADQUARTERS: 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001. (408) 438-530
OFFICES IN: AUSTRALIA, DENMARK, FRANCE, GERMANY, ITALY, JAPAN, NEW ZEALAND, SINGAPORE, SWEDEN AND THE UNITED KINDOM * ARTIKEL-NR. 167302